Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 26
Transactions and lock-free primitives, read/write locks

Welcome to Operating Systems lecture 26, alright.

(Refer Slide Time: 00:29)

So, today I am going to talk about Transactions. So, you have seen locks as a way of providing atomicity and transactions are another way of providing atomicity right. What is atomicity? Atomicity is that you want certain piece of code to appear indivisible right, it should not be possible to interleave between instructions of that code and that code is called a critical section right. And, we use locks to basically you know we made an acquire before the critical section and made a release after the critical section and we said that that will ensure atomicity.

We also studied monitors, we said you know locks are error prone if there is a first class support in the language then it will it become slightly easier for the programmer to reason about things and monitors was basically something that was also providing atomicity, but internally monitors are also using locks right. So, internally the monitors are also basically using an implicit monitor lock to basically provide mutual exclusion right.

So, how do the locks work? Locks work by, you know the programmer has to a priory tell that I am going to access this shared data and he tells it by calling this function called acquire and he calls it on this lock which corresponds for that shared data right. So, it is an a priory declaration by the programmer that I am going to access some shared data and this is the shared data I am going access and the semantics are that the acquirer will not let more than 2 threads access the shared data we will not let more than 2 threads fall right. This can be called a pessimistic way of doing controlling concurrency.

In the sense it is like saying that let us say I wanted to enter a room where I needed some privacy. I always whenever I enter the room, I put a lock inside it before I start doing anything right let us say you know whatever. So, I always put a lock irrespective of whether it is somebody else is actually going to try to enter the room or not right. In the if in the common case it is the case that nobody else is going to come into this room anyways right I am just doing this extra work of putting the lock first then doing my work and then unlocking it and then going out right.

But you know if nobody actually came during this duration while I was using the room you may say that this work is sort of wasteful. And, if this is the common case which in many cases it is right in many cases it you do not expect the probability that somebody will actually access the same resource at the same time is often small. So, you are saying that you are you know you are doing this extra work every time even though the probability of this event happening is small right. A more optimistic method would have been that I do not use a lot, I just enter the room and I start using it and if somebody comes later he opens the room and he tries to use that he figures out that oh somebody else is already using it and then he rolls back his execution.

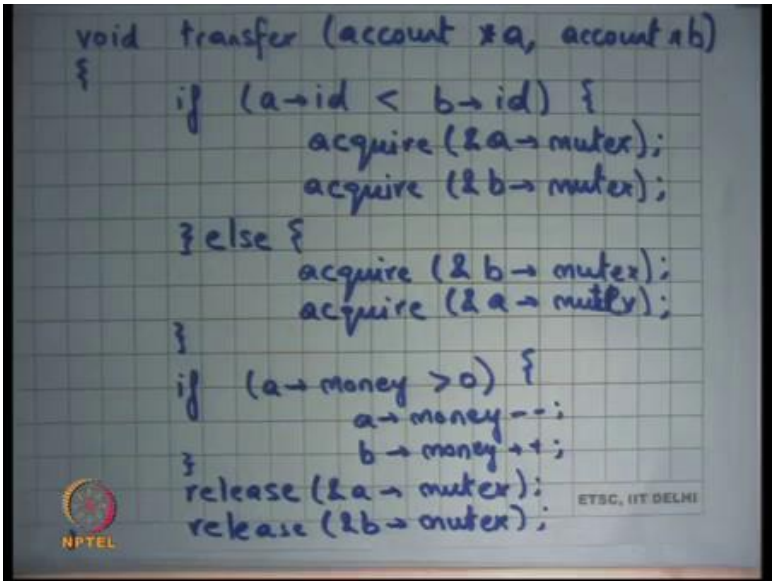
In other words, he just you know in the physical analogy he just goes back and says you know let me ill retry some other time right now somebody else is using it. If I have that kind of an organization then in the common case when there is there is very little chance that somebody else is going to come at the same time you know I did not have to do any extra work to actually put the lock on the door I could just come in do my work and get out and if somebody else comes in during that time hell roll back if needed right.

This is more optimistic, this is more optimistic I call this optimistic, because I am optimistic that the probability that there will be a collision or there will be a concurrency

violation or atomicity violation is small right. So, this is more it is more optimistic in that sense plus locks are more pessimistic there you say that the you in sense you are some in some sense saying that the probability that somebody will actually violate atomicity is very large usually you are always basically doing this right.

So, in software this act of actually coming into the room checking if somebody just is using and if not and rolling back can be done by enclosing some the code into a transaction right. So, this whole process of actually chick coming in and trying to do something and then figuring out that you have failed because somebody else is using it and going back can be done by enclosing this entire code inside what is called a transaction right alright.

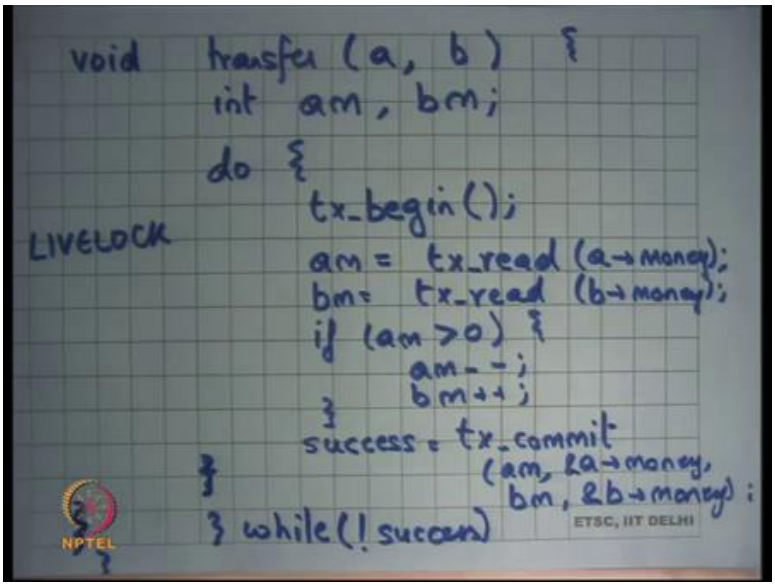
(Refer Slide Time: 04:51)



So, let us look at some examples of how what a transaction look. This example this running example that we have been using for transferring a unit of money from account to account b. And, here is the code which uses fine grained locking to ensure atomicity of this particular function right and we have seen this before.

So, you here is a mutex associated with a as a mutex associated with b and you basically take both mutexes before you do anything and then you release both. Once again this is a pessimistic I call this pessimistic because the probability that there is another thread that will access one of these two accounts a and b is very small right at the same time, yet I am pessimistically taking these locks each time right.

(Refer Slide Time: 05:45)



Let us see how I would have written this in the transactional way right. So, let us see I would have done something like this let us say void transfer from account a to b. I would declare 2 local variables am and bm rep representing a's money and b's money and I will use a do while loop to keep retrying let us. Let me write this code fully before we start discussing it tx_begin() we will say that a transaction has begun, am is a local variable which will read called tx_read(a.money) right and let us say bm = tx_read(b.money).

And then you know so, what I have done is I said that there is a transaction; the transaction will first read the value shared value. So, a.money and b.money are shared values and I am reading those shared values into local variables right and then I am going to perform an operation on the local variables. For example, I will say if am > 0 then am-- and bm++ right and finally, I will try to commit the transaction.

And, in the commit let us say I can give arguments would say right a m the value am to a.money and the value bm to b.money right and I do this while loop till I succeed alright. So, let us see what is happening. Firstly, I declared that what I am going to do, next is a transaction, by saying that everything that I am going to do next to the transaction. It basically means that whatever I am going to do is going to have a tentative effect, it is not going to be a final effect what I am going to do is tentative execution, it is like I am I have opened the room and I am trying to enter it. But, what I am doing it start tentative

execution and if I figured out that there is some collision somebody else was using the same room at the same time then you know I know where to go back.

I go back to the begin point right. So, that is what the tx begin is telling me that here is the transaction and what I am whatever I am going to do from here on is all tentative execution and it is going to be tentative execution till you commit it right. So, this code between the begin and the commit is all tentative execution and the commit function will atomically commit the results of the tentative execution on to concrete state right.

So, what am I doing is I read? So, I started a transaction then I said read the shared value into a local variable, read the shared value in the local variable perform some operation on the local variables and then try to commit the new values or the local variables to the shared variables, this commit function is atomic. So, commit function is atomic it will you know in one go try to update the values of a.money and b.money and it will succeed if between the tx_begin() and tx_commit() nobody else changed or read the value of these shared variables right.

So, I am basically saying here are the shared variables I have touched and here are the new values of these shared variables, commit them to the real state if nobody else has read or written these variables during this time. So, the question is what the problem is if somebody has only read it and I tried to commit it should commit fail if somebody has only read it.

Student: Yes.

Yes, it should right because it will again violate atomicity somebody has read the old value. So, atomicity means that whole thing is appears as either has having done completely before it or completely after it right. So, if I have read let us say a before this transaction started, before this transaction started, I read a and then after this transaction committed, I read b, then atomicity is violated right.

Student: No, but what if let us say there is another function which just used the balance in the account that is left. So, why should I.

Right I mean. So, you are saying that you know it is possible that you may not need to commit, even if there was a conflicting read you may not need to fail the commit even if there was a conflicting read.

Student: Yeah.

That is true you know there may be some situations where that may be true, but in the general case that is not true. So, let us say there was another transaction that is doing the sum right. So, a running example is another transaction that is doing some right. So, let us say it reads a after that the transaction commits successfully and then it reads b that is a problem right. So, that is a problem because you there is no atomicity between the sum and the re transfer right.

So, you know you are right I mean in some cases you may be able to reason about it and say that you know you do not need to fail the commit if it is only a read because it does not matter my code does not you know semantics would not change it is still remains atomic. But, in the general case right because I mean in the general case you would want to say that if there was a conflicting read or a write then fail the commit and sum is an example in this case alright, ok.

So, it is going to commit it is going to try to commit it in an atomic way and if it tries to commit it and it figures out there was a conflicting read and write in this area while I was executing in this area then it will fail the commit right. So, how does it figure out this area? It just looks at you know it just looks at the time when the transaction was begun and between the commits. So, if in this time there was a conflicting read or write, it shows that there was a problem, you can optimize it further you know you can say that you know if there was a read or write here you may be no not clear etcetera.

But so at the, but at the very high level that is what it means that you basically look at what are the conflicting who are who are writing and reading what locations and then the commit operation is going to check this atomically and then either succeed or fail. If it fails then you just look back and you retry the transaction and you retry it a fresh completely a fresh right, you read new fresh values of a.money and you retry it again yeah there is a question.

Student: Means get a non-success or do one thing gets success.

So, do both sides get not success or does one thread get success it depends on the semantics of your transactional memory system you know either are possible, but let us say you know both of them get no success.

Student: But if both of them not get a success then they will get stuck in this (Refer Time: 13:17).

Right, so, depending on how you have implemented your transactional memory, it is possible you can implement semantics way or one of them is guaranteed to succeed which means there will always be progress and you know a simpler implementation and more efficient implementation may only guarantee we may not even guarantee that. So, it may say that you know it is possible that both of them fail right, in which case both of them are going to roll back and which has this problem of what is called a live lock right.

This is different from a deadlock so, it is called a live lock and live lock basically means that here are two transactions, that are continuously doing work, but none of them is actually making progress. In a deadlock none of the threads are doing any work and neither are they making progress, in a live lock threads are actually doing work, yet they are not making any progress. So, they just fall through try to commit both of them roll back and so on right.

And so, that is a really bad situation to have a live lock, but you know transactions are useful if you expect that concurrency is going to be real and so the probability of this live lock is very low right ok. So, all this sounds much more complicated perhaps than the locks yeah question ok, whenever concurrency occurs would not it always result in live lock, well I mean not necessary it is you know it is just a matter of what should you will happened.

So, in the first case it may happen that you know none of them succeeded, in the second case that may happen that one of them definitely succeeded, in you know the transaction memory system will have ways to tolerate a live lock one is that you know one will always succeed that is you know that is one way of doing it. The other is you know if there is a retry then you know maybe have some delay in the middle. So, there are multiple ways of making sure that live locks become less and less probable in the common case right. But, at in any case there is a lot of overhead to actually doing a roll back and retrying the whole thing again alright.

So, yes you can prevent real live locks in many ways one is you know make sure that your transaction system is basically always making progress at least one of them will successfully commit if there is a conflict or you know you basically make sure that retries are serialized you know the other ways that retries will always get serialized. So, you know if you figured out that something has failed then you try to serialize execution yourself you know. So, those are all ways to do things make sure that live lock does not happen here yeah.

So, what is the advantage of such an implementation I said that no locks have this extra overhead of actually locking and you know it would have been better if I had been more optimistic, but this looks even more complicated right. Why is it more complicated or why is it more expensive I should say? Firstly, we have to roll back, but let us say you know rolling back is a very rear operation because I started with the assumption that the probability of concurrent access to a shared resource is small right.

So, let us say the probability of a rollback is very real right and that is true in many cases, that is true for the for the account example the transfer example that we have taken right the bank account example, so anything else.

Student: Data.

Exactly so, you have to basically track what are the locations that have been read, what are the locations that have you know and access and so on and you know how to do this bookkeeping somewhere right and you know how do you do this bookkeeping. You basically you know for this for every memory access for every memory location and for every memory access you have to store some look at some data that is saying you know whether it was accessed or not and these kinds of things and this seems very expensive, this seems much more expensive than actually doing taking just a lock right. A lock just involved setting a bit 2 0 or 1 right.

So, yes, it is expensive because you need to do extra bookkeeping about what memory locations are accessed. Transactions are or you know what resources are accessed, I should say instead of memory locations because memory location is one of one type of resource, but there could be other types of resources. Transactions are used widely in databases or you know anything that involves disk accesses file systems for example, why because it is easy. So, each read operation in transactions that involve disk

operations involves a disk access a disc access or a disk block access or disc access can only be done in granularity of a block right.

So, what would take a you know you either read a block or write a block and you at disk access is very expensive right it is already milliseconds to access a disk. And so, this extra bookkeeping information about which blocks have been accessed and which blocks have not been accessed is very small in comparison. It can be stored in memory and it is very efficient to store this in memory it is not a big deal right. So, transactions are very useful if this operates the read operations or the access operations themselves are very expensive right, then this extra overhead of actually doing this bookkeeping becomes very small relatively ok.

So, transactions are great for something like databases and they are used in fact, to ensure atomicity of database transactions. The other advantage of transactions is that you do not have to worry about a lot of things, you do not have to worry about fine grained locking, you can put a large area of code into one transaction and it is the runtime system that is keeping track of what you have read and what you have written and whether you want to roll back or not right. So, for example, notice that in this code nowhere am I associating a log per account or anything of that sort.

I have just put the entire transfer function within a within one transaction and it is a run time system that is figuring out whether this transfer and that transfer conflict or not right, by look comparing the read write sets you know what memory locations has he touched or what resources has he touched. And, what resources has he touched I am just comparing it as a run time system that is doing it the job of the programmer becomes much simpler right.

So, if he does not have to do fine grained locking he can put a transaction around the entire code, he does not have to worry about you know figuring out the right locks are placing them in the right place, he does not have to worry about deadlocks that is a big one right. Deadlocks as we have discussed is a big problem because firstly, you are to have a global order on all the locks that is often very difficult to get.

You have to reason about you know what are all the locks that different areas of your code have, different types of locks etcetera and all of them have to be in a global order that is a really a difficult thing to do and it also kills modularity. As we have seen before

it does not allow you to write you to write your code in a modular fashion, because now you have to worry about what kind of locks is this he is going to take or he is going to take etcetera and what operations do I need to be an atomic etcetera right, with transactions anything that I need to be atomic I just put a begin transaction and an end transaction around it and we done it ok.

So, it is very useful in that sense and that is one of the primary reason that transactions have actually been very successful in the database community or database world, because you know having to do locking in databases is actually very very error prone. Apart from being expensive the bigger problem were doing locking in databases it is very very error prone you have to worry about what are all the let us say tables in your database and whether you know you are taking the locks in the right order and things like that. On the other hand, you just put it in a transaction and the database system, or the runtime system will figure it out for you alright.

Student: Sir

Yes.

Student: Sir we have said that commit is atomic. So, for ensuring the atomicity do not we require locks.

Right. So, how commit is implemented, is another thing. So, really depends on your run time system how you will implement commit and the question is do not I require locks to implement commit, you may or may not depending on you know let us say you are doing your two implementing transactions on disc locks, yes you require locks. But these locks are only named in memory locks right and more importantly so and more importantly the critical section is very small of your lock it is just about updating the memory right there right.

So, I mean even if you require locks in the commit it does not you know it does not obviate the advantages of or it does not reduce the advantages of transactions that is the problem if it requires locks right. The programmer still does not have to worry about things right and for databases it is still you know equally expensive or less expensive than using blocks right. Now the question is, do transactions make sense in operating systems? Alright.

So, what is the difference between operating systems and operating systems, these are likely to be memory accesses instead of this block accesses and so, this business of actually doing the bookkeeping has is actually you know bookkeep doing the bookkeeping is likely to become more expensive than actually doing the access right, because you have to do at a bookkeeping on every shared access that you are doing. Also the commit is likely to become very expensive because as I was pointing out you will probably need locks and you know and then needing having to go through all the read list and write lists and performing intersections.

So, for that reason you may say that transactions are not really suitable for operating systems and that is what, that is the reason that so far at least in our transactions are not usually used in operating systems and because you know they are too expensive to be used, lock seem to be cheaper to use an operating system. However, you know given that we are moving in the, we are moving towards multiprocessors and more and more processes and so on.

There is a huge need to make more and more of our code more and more parallel and that has huge software engineering challenges for programmers, to be able to do fine grain locking and all this and it makes your coop program very very highly non-modular right. So, later you know very recent processors like the Intel's Haswell processor actually have hardware support for transactional memory.

So, the idea is that they have instructions called transaction begin and transaction commit alright and using those instructions it is a hardware that will do all the bookkeeping for you within that block of transaction begin and transaction commit. And, when you do the transaction commit it is a hardware that will compare the read and write lists of the two transaction and decide whether the commit was a success or failure right.

The assumption here is that doing these things in hardware is likely to be much faster than having to do these things in software alright and there is evidence to believe that at least for some number of processors it is advantages so, to use transactional memory over locks right. And so, that is the reason that a lot of investment has gone from many companies and you know one example is Intel's Haswell processor which is which is which is actually being available today alright ok.

Student: (Refer Time: 24:42) Hardware support is for (Refer Time: 24:45) transactions on disk lock access or.

No, the hardware support is for transactions on memory accesses alright. So, every so, if you say a transaction begin and then you make a memory access it is a hardware that will do the bookkeeping about what memory accesses you have read or written and when you do a commit it is a hardware that will compare these lists of memory accesses and decide whether the commit is successful or failure right.

And, so that is you know that is the domain of transactional memory I would not go into too much detail about it, but just to just to just to tell you about transactions how they have been so successful in databases and how they have being very seriously looked upon as a potential option of handling concurrency in operating systems by using hardware support alright.

Student: Sir.

Yes questions.

Students: To maintaining the modular I think cannot we use conditioner locks such that if it is taking the a function and making a sir function call function call and it (Refer Time: 25:47) the lock and give the lock to the calling function.

So the question is you know I am saying that locks actually hamper modularity and I am saying that because if there is a function that takes a log and then he calls another function that takes another log then there is a possibility of deadlock and all that and the question is if you take one another lock after holding one lock cannot you do something called a conditional lock were the second lock basically is least if you take the second first the first lock is released before you take the second log what do you think.

Student: But the caller would not reach the (Refer Time: 26:21).

I mean if you can just release the lock arbitrarily right I mean the lock is there for a reason you cannot just I mean if you just started releasing locks then you are killing then you are then you are losing atomicity.

Student: But when the function that, sir the function that we are calling it at the same time acquiring a same lock right.

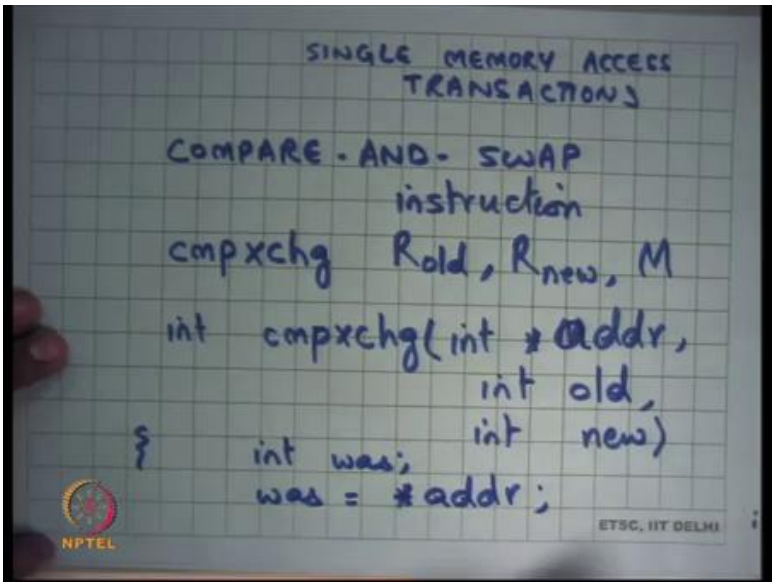
It is not acquiring the same lock it is you know so, modularity is killed because it may be acquiring a different lock right and then you have to worry about the order of these two different locks and I mean if it is the same lock then you know what you are saying is similar to recursive locks that we have discussed.

Student: Yeah.

But and we also discussed problems with recursive locks in any case recursive locks are an option, but really I am not talking about modularity being hampered because of acquisition of the same lock I am talking about modularity being hampered because of acquisition of different locks, because locks need to be acquired in a certain order alright

So, basically transactions are an interesting idea people are looking at it and whether they can be used in operating systems or not remains to be same. However, one type of transaction is actually being used for many years now in operating systems and these are transactions that involve single memory access right.

(Refer Slide Time: 27:41)



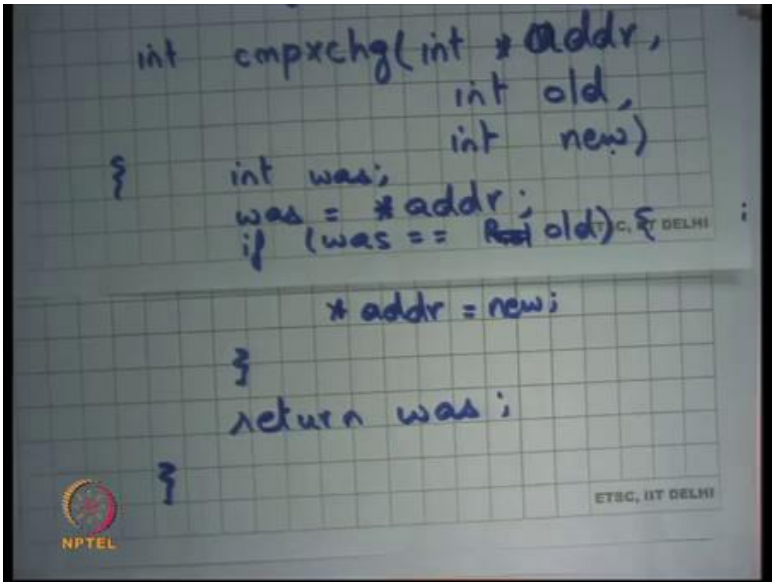
So, if you can write your code as a transaction such that it is going to access only one shared memory location and then you know it is then the hardware has been providing support for a long time to implement such a single memory access transaction. And,

operating systems have been using it for a long time operating systems and in general concurrent programs have been using it for a long time to do this and one common paradigm to do this is basically what is called a compare - and - swap instructions right.

So, let us see what this compare, and swap instruction is if I was to right the c semantic. So, compare and swap on x 86 is called the compare and exchange instruction, it takes 3 arguments register one which I am going to call R old let us say, just say 2 I am going to call R new and a memory location M right. And, the semantics of this instruction are the following I am going to write C code to describe the semantics let us you know let us say I write it as a function then it is compare and swap and star M or let us say address just to make it clear int old and int new right.

And it just no first reads let us say was is equal to star. So, it reads the value in addr into a local variable or locally into the in the hardware for the instruction.

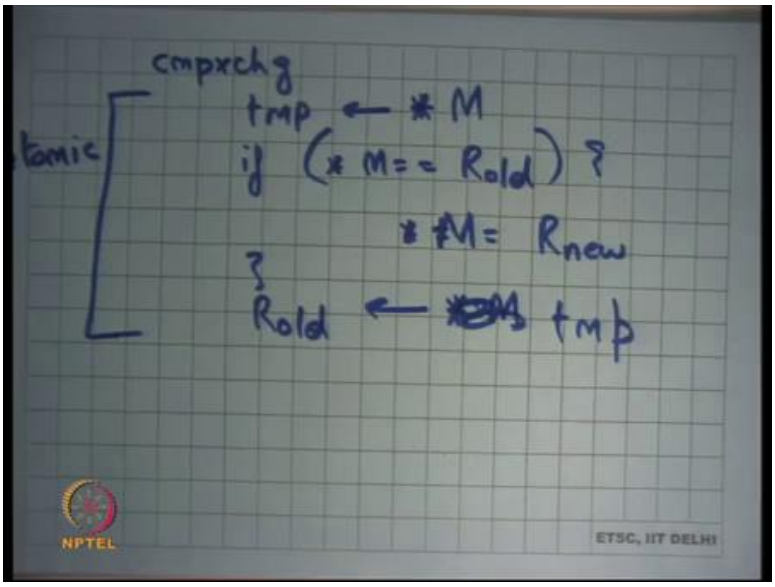
(Refer Slide Time: 29:57)



And, compares the current value with R old or with old let us say in the c things, let us see if it was equal old then *addr = new and return was alright I am going to explain it very soon ok. So, what is going on? Basically the semantics are check the contents at location addr, compare them with location old with value old right, check the contents location addr, compare the contents with value old, if they are equal then replace the contents of addr with new otherwise do not do anything right.

So, the idea is I have previously read some value from this location called addr, I have it in a register. Now, this instruction is going to atomically check if the value of that location is still the same and if so, then it is going to replace it with the new value and if it is not still the same then it is not going to replace it with the new value right.

(Refer Slide Time: 31:29)



So, let me write it in plain English first, compare exchange if *M = R_{old} then *M = R_{new} that is all in some sense right and this is done in an atomic fashion, you check if the value in the at that address is the same as the value that you have in your resistor. And if so, then you replace it with this new value otherwise you do not do anything let us say and just one more thing it basically also returns the old value of *M or let us say let us say temp is *M it returns the old value of *M in R_{old}.

This operation is actually very similar to the semantics of the commit operation in a transaction right. What is the committee operation? The committee operation says change the value of these locations with these new values if nobody has written in the middle right. So, they compare an exchange in suction is the same similar, change the value of this location to this new value if it is value is the same as the value that I have read before right.

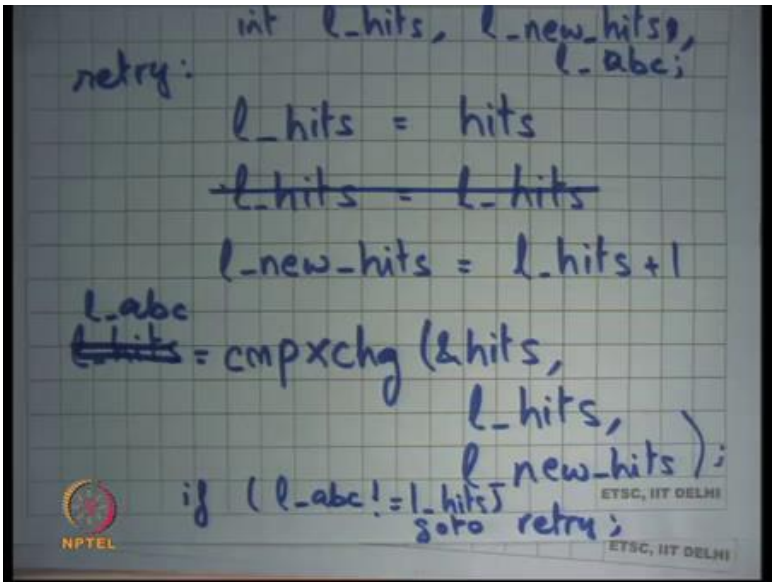
So, there is nobody has written in the middle is captured by the value is the same right and this is an atomic instruction you know you can put the lock prefix behind it to make it atomic.

So, what am I going to do? I am going to use this instruction to commit one memory access to transactions. So, any transaction that can be that has done only that needs to do only one memory access, one shared memory access can be modelled as a transaction. So, any atomic region, any critical section that can be modelled as one update to a shared memory access region can be modelled as a transaction using the compare and exchange instruction right ok, let us see these examples.

(Refer Slide Time: 33:59)

So, let us say I had this operation where I wanted to do hits is equal to hits plus 1 right. So far, I was using locks to do atomicity of the code right. So, I was basically saying acquire here and release here right, now let us try to do this using it transactions right. So, I can say so, if I want to do a hits = hits + 1 and I want to do it in an atomic way I could do something like this I could say.

(Refer Slide Time: 34:33)



Let us say I declare a local variable called local hits and I will declare another local variable called local new_hits let us say just 2 local variables it does not matter how many local variables are declared and I read the value of hits into local hits right. Then I do this operation hits = hits + 1 well let us say sorry let us I do l_new_hits = l_hits + 1 ok.

Now, what have I done? I have read the old value of this shared variable hits in a local variable, I have performed some computation on this local variable and now I have both the old value of that shared variable and what are the new value that I want to put in the shared variable. And so, what I am going to do is, I am going to say compare exchange what, hits as the address, l underscore hits is the old value and l underscore new hits is the new value alright.

So, I am going to atomically try to update the shared variable hits with l_new_hits, but I will only do it if the hits variable still is equal to the old value. If it is not equal to the old value what does it mean? Somebody has concurrently tried to increment hits and so, you should not commit you should not update now right. So, you should not have the lost increment problem right.

So, this will you know this will return the old value of hits whatever the value was read into hits and let us say it is returned and register I want to write it in the functional form. So, l underscore hits is equal to compare an exchange this and I basically say if or let us

say what let us say let us say I say let us say I declared another variable called local. Let us say local abc I do not I cannot so, let us say I just say l_abc is equal to compare exchange this. So, this is going to return the value that was read by this atomic instruction and recall that this entire operation of reading the value, comparing it, and updating it is completely atomic right. So, I basically read the old value into l_abc and I say that if l_abc != l_hits, then what do I do.

Student: Retry.

Retry right so go to retry, now let us say retry is going to be where right you can write it in loop form I am just using go to because of lack of space, but you know basically you just want to retry the transaction you can write it as a wild loop for example, also it does not matter. So, this code is ensuring the atomicity of hits++, but in a transactional way ok, also this transaction guarantees that at least one transaction will always succeed it guarantees progress.

So, this transactional system so, we had this discussion on if there was a conflict or if there was a failure at commit do both roll back or does or one of them will always succeed, in this case one of them will always succeed right. Whoever executed the compare and exchange first and notice that the comparing changes an atomic instruction is going to succeed the second one is going to fail, and he is going to go to retry.

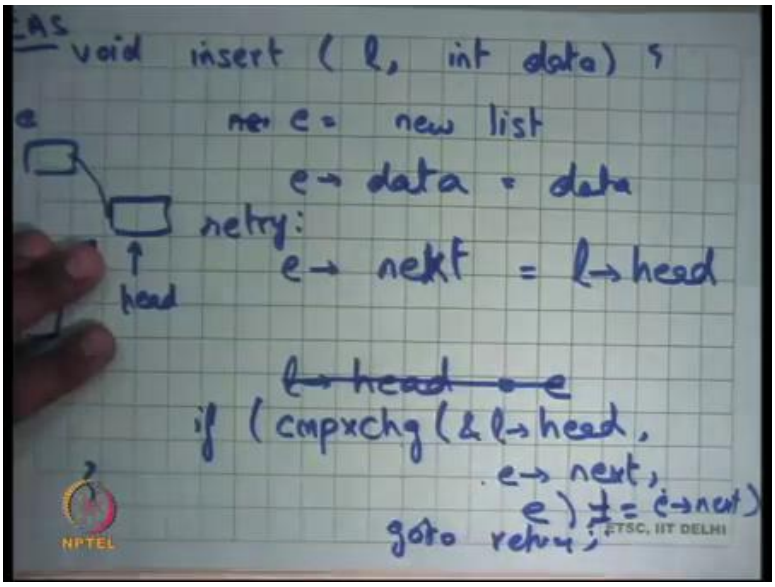
Student: If anybody else reads it would not fail.

Yeah, it will allow multiple reads, so it is basically guaranteeing atomicity against concurrent rights. In fact, it is even weaker than that it is not guaranteeing atomicity against concurrent rights it is guaranteeing atomicity against concurrent changes, if a concurrent right happened, but it left the same value it does not matter right. So, it is not a complete transaction in the strict I mean it is not the same semantics that we discussed it is not comparing the read write sets. It is actually looking at the value and if the value has changed then it will fail if the value has not changed then it will succeed.

So, if there was a concurrent right that you know that wrote it, but it remained the value remain same no problem right. So, that is the semantic of (Refer Time: 39:12) and whether it fits in your logic program logic or not that is really up to you to decide ok. So, notice that I have avoided locks in doing so, and so all the problems associated with

locks are also gone in some sense. Firstly, I do not need an extra shared variable which is a lock and secondly, you know because there is progress guaranteed now there is no problem of deadlock alright. Now, let us take another example we were looking at this insert function right initially.

(Refer Slide Time: 39:43)



So, we said insert into list l some data right and we had this lock we had this peerless lock to do this insertion and let us try to write this insert as a transaction right. So, how let us see I mean using comparison swap right. So, comparing swap is also called CAS and I am going to use the word CAS to represent compare and swap instruction or compare in exchange right, it is a common term called CAS to use to basically do this right.

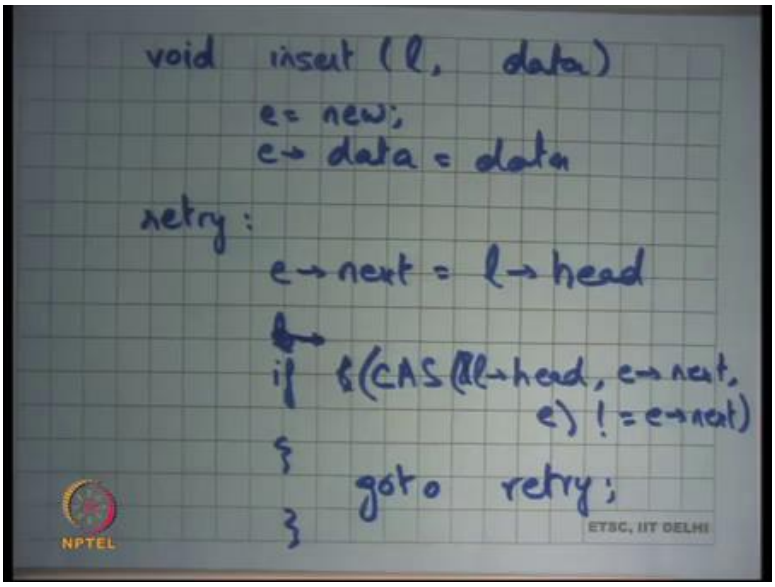
So, how can I do this, well I can say let us see new let us say e is equal to new list, e.data = data and then I say e.next = list.head right and finally, I am going to say list dot head is equally right recall that was my code list dot head is equal to e right. So, notice that all these 3 locations or 3 instructions are only operating on local data or at least read writing to local data they are not writing to shared data.

It is only this final instruction that is writing to share data right and it is possible to write this using CAS without having to use locks in the following way, you can say you can do compare an exchange on l.head with old value being e.next right and the new value being e good right. So, you atomically try so, what is happening let us say this was head you

have already made e and you have made it point to head and now you atomically try and you already know that you know the time when you read l dot head, head was here. Now, you automatically tried to make head here, but you only do that if head is still equal to this. If there was a concurrent insert, then head would have become something else right if there was a concrete insert head would have gone here.

So, let us say there is a concrete insert and somebody was trying to do this and so, let us say he won then head would have come here and so, the second insert would have failed right. So, if this succeeds that basically means that nobody has been able to do a concrete insert for me and so, what I do is, if you know if compare exchange is equal to e dot next. It is not equal to e dot next then retry right so, go to retry, and retry is again, where should retry b yeah just before here right let me write it again.

(Refer Slide Time: 43:01)

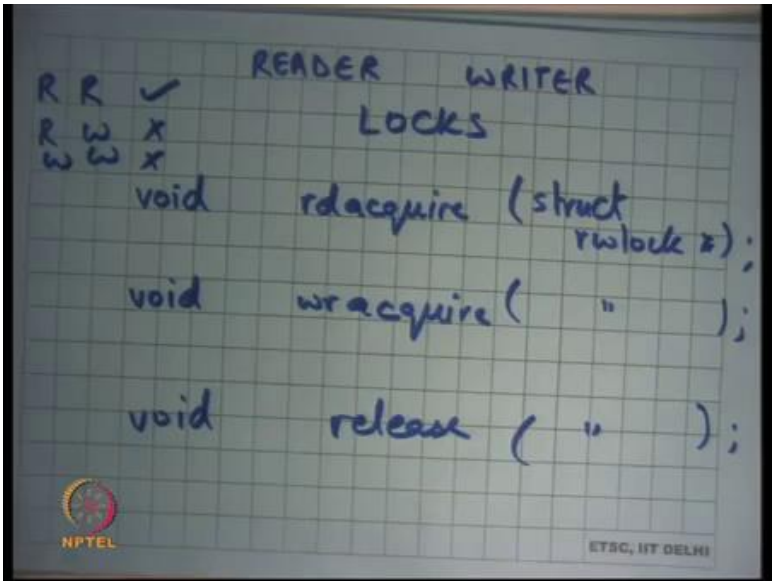


void insert (l, data) e = new; e.data = data, retry e.next = l.head; l.head or now this is the this is the tricky part if CAS instead of compare and exchange I will just use the word CAS if (CAS l.head actually &a.head, e.next, e) != e.next then go to retry ok. So, this is a lock free way of ensuring the toxicity of the insert operation alright.

So, these are these are called lottery operations it is compare and swap kind of things are called lock free operations and you know at the heart of it is really a transaction with a single memory location alright, with the different kinds of semantics it is not doing read write set intersections it is actually looking at the value to decide whether there was a

conflict or not. Finally, so, any questions on transactions before I move to my next topic very briefly, alright good.

(Refer Slide Time: 44:51)



So, finally, I like to point out that there is something called a reader writer lock right, the idea is that often they are multiple functions, some of the functions are only interested in reading the shared memory location or shared object and some location some functions are interested in also writing to the shared memory location.

It should be possible for multiple threads to concurrently read because you know one read does not you know adversely affect another read and it is only a write that effects adversely effects a read or it is a only a write that adversely affect a write right. So, the conflicts are only between read writes reads and writes or writes and writes there is never a conflict between a read and read.

So, if you expect that your code has lots of readers and very few writers then it is possible to use this change of fraction called reader writer locks. And, it has 3 types of functions locks have acquire and release these have read acquire and read release and write acquire and release right same struct lock, let us say struct r w lock right. So, the idea is that the fraction now says that multiple threads can hold the lock in read mode simultaneously or the lock can be held in the read mode simultaneously, but if somebody hold, but a lock can only be held once in write mode alright.

So, acquire basically made sure that only one the lock can only be acquired once at one time right it cannot lock cannot be acquired by 2 threads simultaneously. In this case a lock can be acquired simultaneously by 2 threads in read mode that is possible. But if, but it is not possible to have a lock being acquired in by 2 threads in write mode, it is also not possible for a lock being acquired in a write mode and the read mode simultaneously right.

So, in other words read and read, read is allowed, but read write or write, write is not allowed ok. So, that allows you to have more conquering a system if you expect that they are going to be lots of readers and very few writers and first alright good.

So, let us stop.