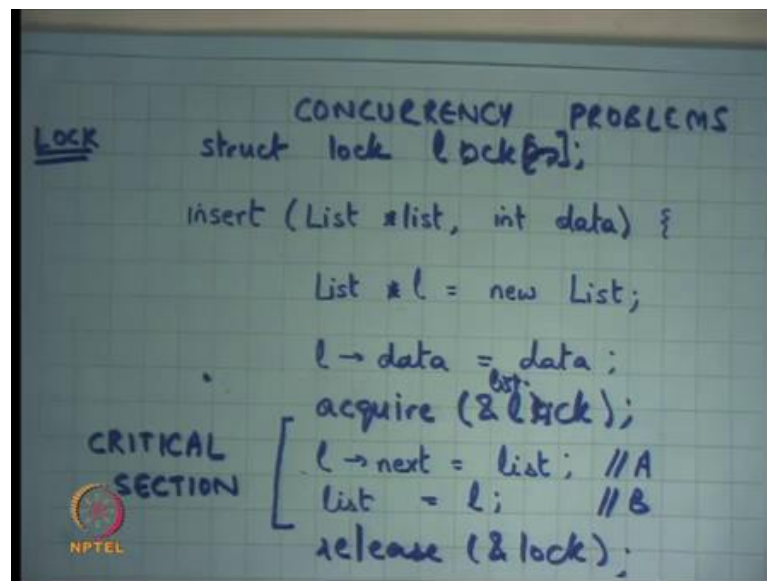


Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 21
Locking

Welcome to Operating Systems lecture 21 all right.

(Refer Slide Time 00:29)

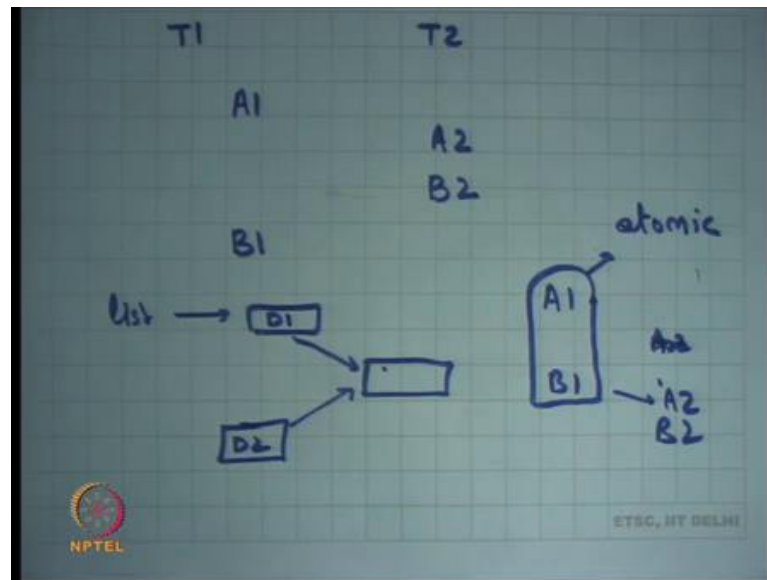


So, last time we were looking at concurrency problems right. And, we said let us we looked at this code, which inserts an element data into a list right and let us say this is a link list. So, the way this works is you declare a local variable, you allocate some node of type list from the heap, you said it is data, you said it is next pointer to the list, the list that exists that is even as the first argument and you update the list right.

So, you are basically upending the data in the front of the list. Now, and we said that look these two statements let us call this statement A and let us call this statement B; these two statements are needed to execute in an atomic fashion. So, they while the executing they should not get interrupted or nobody else should be touching this list while in the middle.

So, it should not happen that in the middle of this and other thread starts executing insert, if it does that then bad things can happen right. And, the bad thing in this case would be that the list structure is no longer preserved.

(Refer Slide Time 01:57)



So, for example, we saw last time that if it so, happens that thread the two threads T 1 and T 2, one thread executes A 1, the other thread executes A 2 and let us say it executes B 2 after that and then this one executes B 1 right, then what can happen, you had a list and thread one will insert it is data, let us say this is D 1 data 1.

And, thread two will insert it is data to the same list right. And, you will have something like this. So, list will be pointing here, and this is no longer a list now right. So, bad things like these can happen. Similarly, you know any other schedule which you know A 1 A 2, A 1 and D 1 getting interrupted and another insert getting called on that list bad things can happen.

So, another bad schedule is let us say A 1 A 2 B 1 B 2 all right or anything on this one. So, these are all bad schedules, and this basically corrupt your data structure right or violate your invariance right. So, what did you want, you really wanted that you know A 1 and B 1 should execute in a sort of non-preemptable way right? So, or in an atomic way ok.

So, if it is possible that while A 1 and B 1 are executing, now either you know no other call to insert should start or no other call to A and B should start. So, A 2 should either be here, or A 2 and B 2 should have been before A 1 and B 1 right that is what you want ok, all right.

So, in other words you want to say that these two statements A and B should execute atomically, and they should execute atomically with respect to something with respect to themselves right. So, they can be other code that is running simultaneously, but as long as that code is not touching this list it is right, but if that code is likely to touch this list, then it should not be now that code in this code should be atomic with respect to each other all right.

And, so we said that there is a there is an abstraction called a lock right, that we are going to discuss today. So, what are some so, when we are designing such an abstraction, what do we need to take care of firstly, in this function these are the only two statements that need to be atomic right, it does not matter what happens here right. So, it is only A and B that you need to protect.

So, you need some way of specifying that this is the area that is atomic, you do not need to make the entire function atomic number 1 right. Number 2, it will be nice if you could say that you know operations on this list need to be atomic with respect to each other, but operations on list 1 and operations on list 2 do not need to be atomic with respect to each other. So, you know insert on list 1 can happen concurrently with operations on insert on list 2 it does not matter right.

So, you would like to add that kind of flexibility, that allows you know better concurrency and better performance in general right. And, thirdly it may not be just this function insert right. So, insert it should be atomic with respect to each other or these two statements should be atomic with respect to themselves, but these two statements may need to be atomic with respect to some other statements right, but let us say there is another function called delete right.

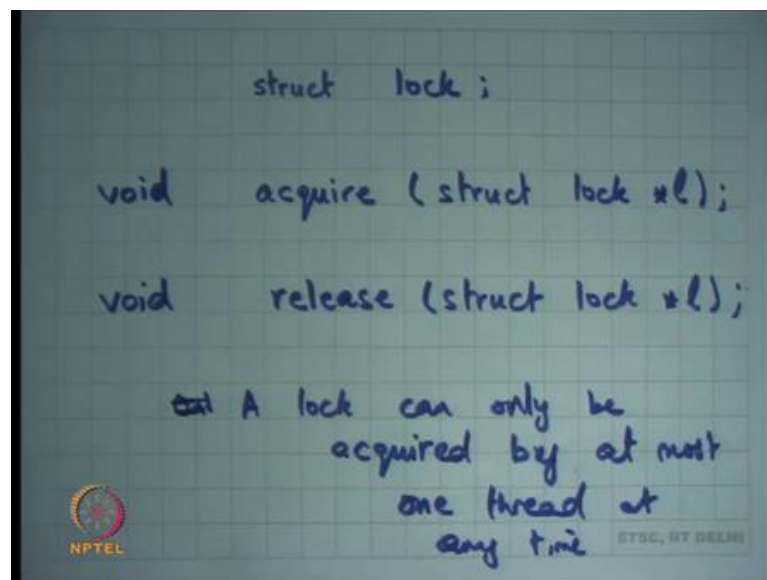
So, delete and insert are happening concurrently, then say similar situations can happen where the list becomes corrupted right. So, it is not just these statements being atomic with respect to themselves these statements may need to be atomic with respect to some

other coordinational system right. So, that is you know that is the so one has to designing abstraction that allows you all these different flexibilities right.

So, many different abstractions exist to solve this problem all right. And, you know as you go along in this course or in future courses you are going to come across, different kinds of distractions, different programming languages have different abstractions you know C will have a different kind of.

So, Java or something like that would have different kinds of support and different languages have different kind of support but locking or locks are one of the most common types of abstractions used to solve this problem all right.

(Refer Slide Time 06:33)



So, what is the lock? All right. So, a lock is defined by a type let us say the title struct lock right, or any other type there should be a type it says it is a lock right, and there should be 2 functions defined on this lock; one is called acquire right, which basically represents that the lock has been acquired all right. So, I am going to discuss what that means, and release all right.

So, this variable, this structure lock is a stateful structure and it represents two states; either it is locked or it is unlocked, or the other way to let us say it is either it is required or it is released right. So, that is the you know so, lock has 1-bit state of 1-bit state, which says whether it is locked or not.

The function acquire changes the state of that lock from unlocked to locked, but the precondition being that the state should have been unlocked earlier right. So, if it is already locked then acquire will just wait till it becomes unlocked and as soon as it becomes unlocked it will turn into locked all right. So, that is the semantics acquire once again.

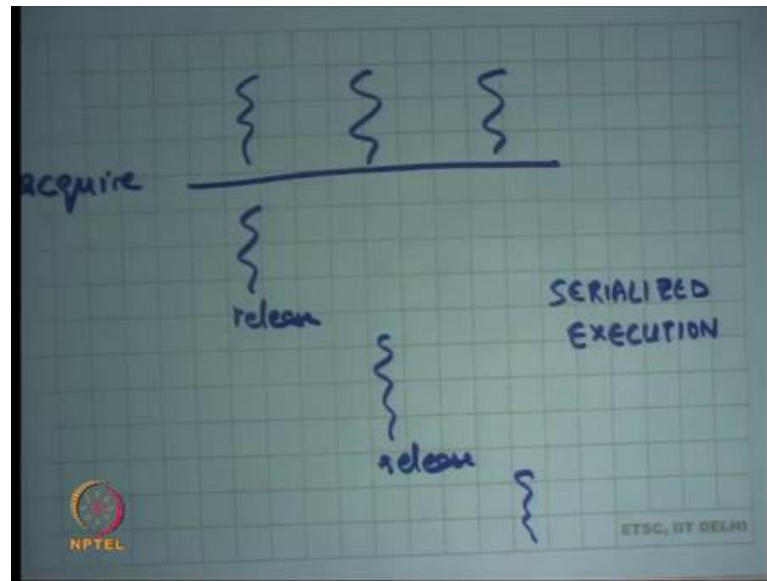
Acquire will set this state of this lock to acquired state or locked state, but it requires that earlier it should have been in the unlocked state or release state right. If, it is already locked then it will keep waiting till it becomes unlocked and after it has become unlocked it will set it to locked right. And, release will just make set it to unlock all right. So, that is basically it all right.

And, let us assume that these functions are you know are implemented in some way, how they are implemented we are going to discuss this later, but so, what happens is if two threads try to acquire the same lock simultaneously and let us say initially the lock was unlocked. So, let us say initially the lock is unlocked and two threads try to acquire this lock simultaneously only one of them will be able to lock it and the second one will keep waiting all right. So, that is abstraction.

The second one will keep waiting till what time till the first one calls release all right. So, in other words there is an invariant, the invariant is only a lock can only be acquired by at most one thread at any time. It is only one thread could have acquired the lock at any time; the second thread will have to wait.

Only out of the first let us say there are 5 threads or n threads then other $n - 1$ threads will have to wait, only when the first thread calls release or there is somebody else going to get a chance to get the lock all right. So, what is it doing? It is basically doing it is if multiple threads call acquires, then it is serializing these threads, it is saying multiple threads have try to call acquire.

(Refer Slide Time 10:14)



So, if multiple threads let us say call acquire and let us say they come at the same acquire point. From A 1 let us say only 1 thread will get to run, only when it calls release then the other thread will get to run, when it calls release the third thread is going to run right. So, that is serialized the execution, but in a serialized execution only for the region which was within that acquire and release calls all right.

So, let us see what will happen in this case? So, we had this code, we had this function called insert that insert data into a list, and we want we were not happy with the fact that this code could be executed concurrently by multiple threads. So, what do you want? You want to serialize the execution of this code right.

So, one way to do that is define a lock all right. Define a lock and initially let us say the state of the lock is unlock and just put lock calls to acquire the lock and release the lock right. I think there is let us say let us call it lock. So, acquire lock and release lock all right.

So, this makes the code correct right, because if there is one thread, it will come here, the first thread will be able to acquire it, violated the inside this and other thread who tries to execute the same code will not be able to acquire the lock. So, it will have to wait. So, we have disabled concurrency in this region right.

So, this region which is protected by locks is also called the critical section right. So, critical section is a region of code that needs to be serialized and its execution and the locks are basically allowing it to get serialized yes question.

Student: Concurrent threads which you are talking they are on different CPUs or they can (Refer Time: 12:40) same CPUs.

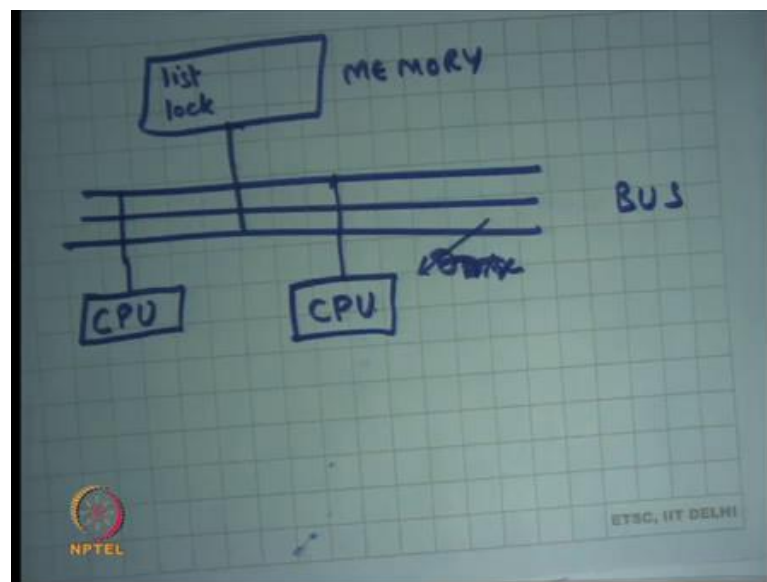
Right good question. So, what are these concurrent threads that is which I am talking about, are these on the same CPU or are these on different CPUs?

Student: (Refer Time: 12:49) same [vocalized- noise].

Same 6 ok, 10, 15 people different right, actually it does not matter all right. It can be on same CPU it can be a different CPU, let us look at how threads work right. So

Student: Sir, if is on the same CPU and if suppose I have acquired a lock and then it gets (Refer Time: 13:12).

(Refer Slide Time 13:20)



All right. So, let us look at what happens? Let us say so, let us say there is one CPU right, and let me just draw the diagram that we have, this is the bus and let us say this is the memory right. In the memory there lives a list right and what is happening was that this CPU was trying to update the list right.

And, while it was in the middle of updating the list it is possible that a timer interrupt comes right and the thread that was executing gets preempted or it gets switched out. And, another thread gets to run in which case what will happen is while I was executing at this line, an interrupt comes I get context switched out the other thread gets to run and have the same problem as earlier right. So, this problem exists even with one CPU.

If you have multiple CPUs the problem still exists it may be it becomes a little bit now the probability becomes higher, because it is possible that one thread is executing here and another thread is executing on the CPU, they are both trying now there is no interrupt involved right. So, there is no interrupt involved, but both these threads are trying to simultaneously access list and the same thing. So, same problem can happen.

So, the problem is really about interleaving. The interleaving can happen either, because of switching out on the same CPU or the interleaving can happen either because of physical concurrency of physical CPUs on physical memory right. In either case the nature of the problem is in the same all right. I have a question for you. Does the problem exist for user level threads or does it only exist for kernel level threads?

Student: Both (Refer Time: 15:07).

It exists for both right it does not matter whether it is user level thread or a kernel level thread it exists for both, because at the end of the day the abstraction is that the thread can be switched out anytime and another pair can get to run. At every instruction boundary a thread is potentially switchable all right. So, it does not matter ok.

Student: Sir.

All right question.

Student: Sir, suppose we for example, you have given in case of multiple CPUs, sir it does this lock will say like stop all the other threads running in all other CPU is adjust.

So, now, so let us say what happens with the lock right. So, we said there is a list living in memory. Now, in our new abstraction we also have this extra variable called lock living in memory all right. And, now one of these both these CPUs if both these CPUs try to acquire the lock in the same time, somehow this acquire function has been

implemented such that only one of them will succeed the other one will start have to wait right.

How it is implemented we are going to discuss in a minute? But, let us just say that there is this magic function called acquire that allows us to do it all right. So, assuming that there is such a function you know life becomes easier, I can put a lock around my critical section all right.

In fact, notice that this abstraction allows you to say that this is the critical section I do not need to say the entire function is a critical section. So, I can you know, I can just protect this area and these lines can execute concurrently. So, I do not need to you know impede performance on the other lines.

The other thing is if let us say there is another function called delete and delete is going to do some manipulations on the list also, then what you need to do wherever you doing those manipulations you need to use the same lock there right, you will make sure that you using the same lock in both in certainly. So, that way you can prevent you can make sections of code atomic with respect to each other by using the same lock around those section those critical sections right.

Thirdly, the third thing I said was I may I want to say that look this list and this list do not need to be mutual do not need to be aware of each other, they can proceed concurrently. So, in this case you know am I preventing that so, if I insert in one list and if I insert in another list will they get serialized.

Student: Yes (Refer Time: 17:36).

Yes, right they will get serialized, because a 1 lock and you know I am calling insert on l 1 and I am calling insert on l 2 the same lock is going to get taken and so, only one of them is going to go forward. Even, though the list was different I did not need to serialize, but it will get serialized is it wrong to over serialize.

Student: Yes, Sir.

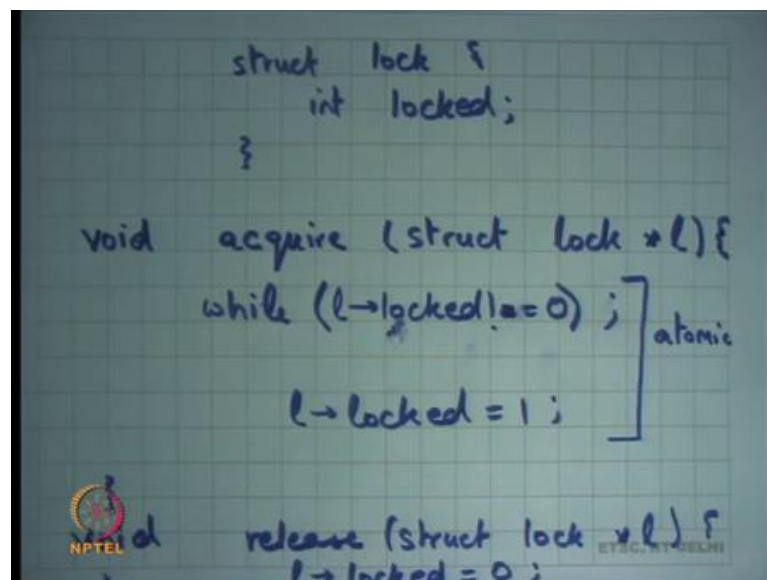
It is not wrong.

Student: The program.

The program remains correct, but performance gets affected right. So, what would having a better way better way would have been let us have a lock per list all right. Let us have you know let us have an array of locks right or better still in the list structure itself I have a lock right. So, instead of saying acquire lock I say acquire list dot lock or something right.

So, in the list structure itself I have a lock. So, this is a lock for this list. So, if you manipulating this list take this lock, we manipulating that list take that lock, but these 2 locks are separate lock. So, you can do this concurrently right. So, that is a thing about lock they give you a lot of flexibility all right ok.

(Refer Slide Time 18:48)



```
struct lock {  
    int locked;  
}  
  
void acquire (struct lock *l) {  
    while (l->locked != 0) ;  
    l->locked = 1 ;  
}  
  
void release (struct lock *l) {  
    l->locked = 0 ;  
}
```

The image shows handwritten code on a grid background. It defines a 'lock' structure with an 'int locked' field. The 'acquire' function uses a while loop to wait until 'l->locked' is 0, then sets it to 1. The 'release' function sets 'l->locked' back to 0. A bracket on the right side of the while loop and the assignment 'l->locked = 1;' is labeled 'atomic'.

Now, let us see how locks are implemented? So, what do we need to implement we need to implement a structure called lock all right? And, let us say I implement it by having a field inside the structure, which says that I am locked or not I said this is state full abstraction, it basically has one bit of state which say that I am locked or not. So, let us say this is the state will lock.

Student: Can we have lock on subset of data structure may be like tree it has a tree and I already want to lock the left structure (Refer Time: 19:15) is it possible sir.

Can we have a lock per subset of a data structure? For example, if I have a tree then I want to only lock one sub tree and not the other side of the tree is it possible, yes, it is

possible ok. How, it is done I mean we are going to see as we go along the course all right.

So, let us see why acquire struct lock star l right. So, I need to implement this was one way to implemented well it is one very simple way just check right. So, just check is l dot locked is equal to 0, if it is 0 then very good just set l dot locked is equal to 1 and return right.

But if it is not 0 then keep waiting. So, let us say to wait I just put a while loop here right. So basically I check if l dot locked is equal 0, if it is 0 then actually let us say not equals 0 right. So, while l dot locked is not equal to 0 I keep spinning as soon as I find it to be equal 0 I said it to locked right, but if it is not equal 0 it means it is already taken I cannot take it I need to wait right. And, the way I am waiting is I am just checking it over and over again in a while loop all right.

And, how do I implement release, I say l dot locked is equal to 0 all right really a simple. I do not need to do any waiting I just need to revert the state back to 0 right. So, that is acquired that is released is this correct.

Student: Sir, but how do we define the preference between functions like, suppose there were delete and insert and suppose locking was occurring.

Ok.

Student: So, it right needs the gate start first.

All right. So, question is how do we define the preference all right? So, preference let us say there are 2 threads one of them is calling insert, the other thread is calling delete we should get first all right. Well, the order can be completely arbitrary right, the lock have actually does not say that you know does not give an order.

In general, you would just from a performance standpoint what is the best policy well one big one big policy is first come first served, whoever comes first given the lock all right. Who whoever come second you know has to wait irrespective of what function you are executing?

Student: Sir, then things will not be predictable or not.

Then things will not be predictable from what happened in science sense of way yes. So, there is still some non-determinism. Even, after you have locks it is possible that this happens right thread 1 executes before thread 2, what is possible that this happens thread 2 execute before thread 1. And, both of them will result in different final values of your memory all right.

So, this is called non-determinism there is still some non-determinism in your system depends on who gets there first, but the thing important thing is it is correct, it is not wrong. Earlier it was wrong, because I was allowing this and this is wrong, it was corrupting my data structure. This was right this was right, and I am allowing both of them and that is they tell some non-determinism in your system right.

In general, it is a very interesting thing that systems need to be non-deterministic for performance right. There is a very fundamental thing, if you make a system very deterministic, you lose performance right. So, let us take an example, let us say I wanted to make my system deterministic.

So, I say you know this lock will always be taken in strict round robin order, for thread 1 will be I know how many threads there are let say there are 2 threads. So, first I will give it to thread 1, then I will give it to thread 2 and so on right thread 1, thread 2, thread 1 thread 2.

So, if thread 2 reaches first, then I will just have to make it wait, even though thread and let us say thread 1 is line behind for whatever reason, there is a thread 1 is executing on a slower CPU, let say thread 1 didn't get enough time to execute while I was doing it. So, you know for all these or you know any other reason, thread 1 or thread 1 in general has more code to execute then thread 2 right all these are possible cases.

In either case the thread 1 is lagging behind thread 2 and thread 2 reaches a lock first if you want to have complete determinism, then you will have to make thread to wait which is just waste right, because there is nobody who is actually executing in that region, you could have executed it correctly you could have led thread to execute first, but you have to make it wait.

So, non-determinism allows performance and lock abstraction in general does not thought nondeterminism. It allows you to provide correctness, but there is still some non-determinism in your system ok.

Student: Is that possible 2 threads are bucking on a same list and 1 thread acquired a lock, but the other thread releases the lock without acquiring it (Refer Time: 24:29).

Is it possible that one thread acquires the lock while the other releases the lock? So, is it possible for a thread to release a lock without acquiring it?

Student: (Refer Time: 24:39).

It is possible, but you know we was assuming that because they are threads, they are trusting mutually trusting each other right. So, it is the programmers mistake that he has you know he is called list without acquiring the lock, it says problem he will pay for it by some error in his program and unless he is doing it very carefully all right.

So, notice that because you are operating the threads environment, we are basically saying that we are trusting other threads. So, threads must acquire and release for put acquire and release around all critical sections appropriately, they should not call acquire twice without releasing one thread should not call acquire twice without releasing, release should not be call without calling acquire you know that is. So, this is these are all invariant that the program I need to maintain. So, the headache is programmers.

Student: Sir, but if the struct lock maintains the like a list of threads, which have acquired it then may be like in the implementation themselves we can check this quality.

Right. So, firstly, a lock cannot be acquired on multiple threads simultaneously right, a lock has to can be acquired only by 1 thread at a time right. So, it is not a list of thread that has acquired the lock it is only 1 thread that could have acquired the lock at a time all right. And, the other thing is when you are using the lock it is better to not rely on the implementation of the lock.

So, that you can change the implementation underneath, just look at the abstraction; the abstraction is there is a structure called lock and there these two functions acquire and release. Do not make assumptions about how they acquire and how the release is implemented? In general, that is how; that is how it is done? So, that you know the

implementation of the abstraction can be changed later at well right. So, this is the standard practice that you will learn the abstraction ok.

Let us come back to this implementation is this implementation correct go ahead.

Student: Sir can I context switch while I am locked like can I.

Can I context switch while I am holding the lock.

Student: Yes.

Yes.

Student: Sir, then the other threads cannot do anything, because since I have locked it.

All right.

Student: It means it is basically wastage of some performance.

So very interesting question, if I have a lock I have taken a lock and in the middle of the critical section, a context switch happens, another thread gets to run, if that thread also tries to take a lock, take the same lock, then we will just have to wait all right. Till the next context switch and there are some wastages of performance absolutely true.

How these things are handled let us talk about it in sometimes. Actually, discussion about whether a context switch can happen, while you have taken a lock or not really depends on your implementation of how you have implemented the lock all right. So, but in any case, it is correct right. So, let us talk about correctness first and then we are going to talk about performance.

Student: Sir, if like thread one sees that locked value 0 and that context switch will take place and the other thread will also see that like the value of locked 0 then.

Good. So, you are talking about a problem in the 6 implementations yes. So, everybody understands that a problem in this implementation.

Student: Yes, Sir.

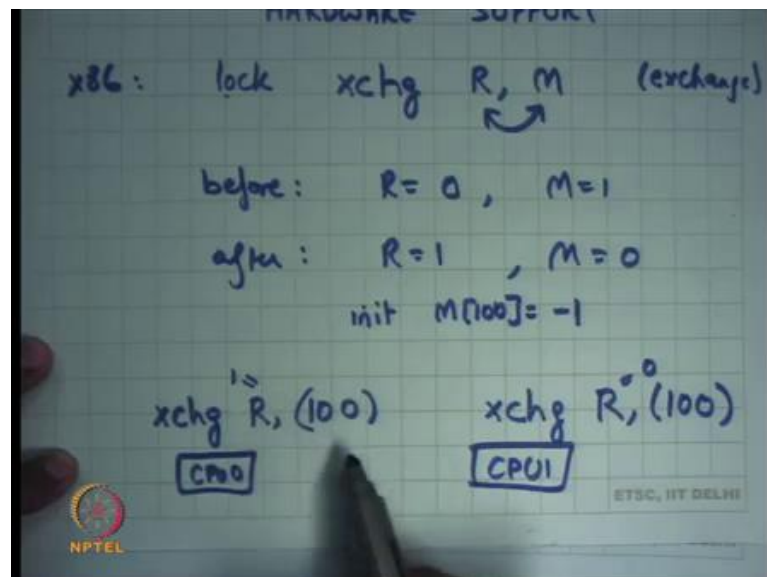
Right ok. The problem is one thread comes here, it sees that the lock is equal to 0 it is here a context switch happens or forget about a context switch let us say it is a multiprocessor system, and other thread also comes here, and both of them are here both of them take the lock you have broken your abstraction right the abstraction is no longer true. So, what did you want?

Student: (Refer Time: 28:05).

You wanted that this entire thing should have been atomic right. So, what am I done? I just you know I had some critical section that I wanted to make atomic, I said I am going to use locks to protect that critical section, but now I have changed the problem to say now I have to implement this lock, but now to implement the lock also I need some atomic away right.

So, I have not really made any progress except I made some progress in the sense that I have reduced the critical section from that large area, which can be arbitrary large to the small thing here all right. And, now the question comes, how does 1 implement this in an atomic fashion right all right.

(Refer Slide Time 28:55)



So, the answer is you need hardware support. So, there should be some instruction in the hardware that allows you to read and write a variable in an atomic fashion right. So, I am

going to take the example of x 86, on the x 86 there is this instruction called exchange all right. An exchange takes 2 operands: a register and a memory address.

So, for example, I could say and so, and the semantics are that it is going to exchange. So, this is exchange instruction solidification and it will exchange the contents of register in memory all right. So, if you know let us say initially so, before R is equal to 0, M is equal to 1, then after you will basically see R is equal to 1 and M is equal to 0 right. So, that is the; that is the semantic of the chain instruction.

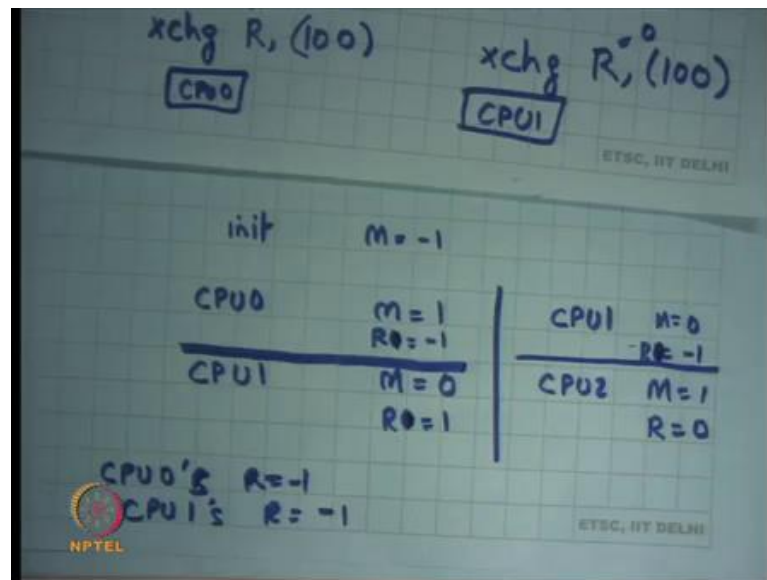
Notice that this instruction is doing two memory accesses in one instruction right. It is reading the old value of M and it is writing the new value of M. So, it is now it is both reading and writing in one instruction right. So, that is important. So, that in that sense this instruction is slightly special because it needs to read and write memory. So, it needs to make 2 memory accesses in some sense all right ok.

And, it is possible to make an instruction atomic in hardware. So, on x 86 there is this lock prefix. So, if you say lock exchange R M then this instruction will atomically read the value of M and overwrite it with the new value, such that no other CPU can intervene in the middle right.

So, in what does it mean that if let us say there is CPU 0 and CPU 1, that vertex execute exchange on address let us say 100 all right and let us say I am saying. So, this is address and this is value let say R and let us say R was equal to 1 and this one is saying exchange to the same address and let us say this was 0, then either this will happen before this or this will happen after this, it is not possible that the two reads and writes of one instruction get interleaved right.

So, for example, right so, let us say initially the 100 th location had value minus 1, then and let say CPU 0 executes before so, minus 1 comes to R and 1 goes to M right. So, this was initial of all right. So, let us look at this again in a bigger context so, all right.

(Refer Slide Time 32:11)



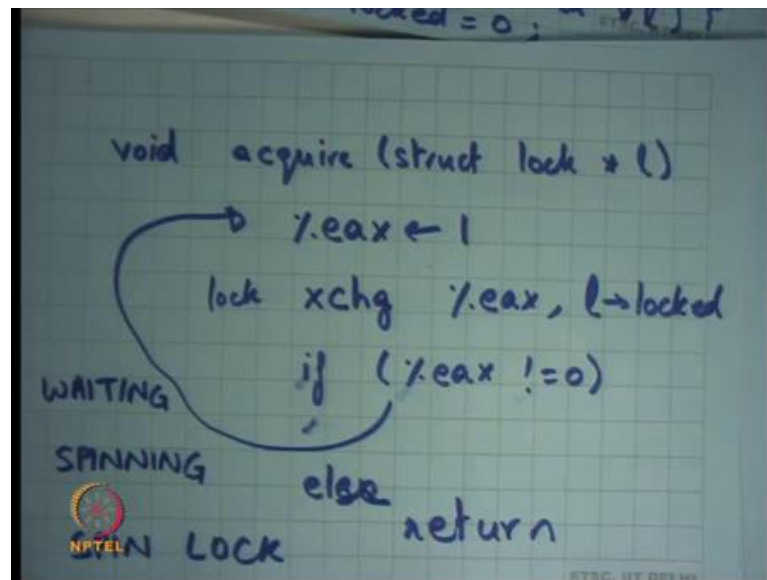
So, let us say initially M is equal to minus 1 if CPU 0 executes then M will become 1 and if CPU 1 executes after that M will become 0 right and so, the final value will be 0. On the other hand, if CPU 1 executes first, then it will become 0, and CPU 2 executes later then comes 1 right. What is not possible is that they get interleaved right. So, what can happen if they get interleaved? Both of them read the old value of M and so, in this case you know CPU so, R 0 would be equal to minus 1 and R 0 will be equal to 1 in case of CPU 1 right.

In this case R 1 will be equal to minus 1 R 0 of CP 1 will be equal to minus 1 that is now called R 0. Let say R is equal to minus 1 R is equal minus 1 R is equal to 0 right. So, these are the only 2 possibilities, if 2 threads 2 CPU sideways execute exchange, they are atomic with respect to each other, what cannot happen is that both of them read the old value of M.

So, it is not possible that CPU 0's R is equal minus 1 and CPU 1's R is also equal to minus 1 not possible right this could have happened. If both of them read old value right and then they try to overwrite the new value and so, in that case exchange intervene atomic with respect to each other right this clear.

Basically, what I am saying is that the exchange operation itself is atomic with respect to itself and with respect to any other operation on that particular memory address all right.

(Refer Slide Time 34:25)



So, with that let us see how I am going to implement acquire, I am going to say void acquire And, I am going to you know use some register let us say I use register eax I am going to put a value called 1 in the eax. Now, I am going to do exchange eax with l dot locked right. Why was the memory address of l dot locked? And I am going to do in atomic fashion so, I use a lock prefix here.

So, what is and then I am going to check if eax is not equal 0 do what spin right else return right. So, I have taken the old value of lock into eax and put the new value of lock eax into lock. So, in 1 instruction in 1 go I have read the whole value and I put the new value into lot. If, the old value was 0 which means it was unlocked then I am done. If, the old value was 1 it means it was already locked. So, I did not change the value of lock either. So, I just keep retrying this is clear.

So, this my register I have set it to 1 I am trying to push this value 1 into locked, but I am; but I am also making sure that the old value of lock should have been 0 right. So, that is what I am checking here. If, it was if the value of lock was not 0, then it basically means that the lock was already locked. So, I keep trying again.

So, what is the difference between the old code and the new code? In the new code I am using this exchange instruction to read and write the value of locked in 1 go right. So, if the 2 threads which are trying to acquire this lock at the same time one of them is going

to hit the exchange instruction first. Over hits it first it is going to get the lock and it is going to return. The second one is going to see the value of locked to be one.

Student: The 1 which cannot acquire a lock will keep on performing exchange operation again and again.

Yes.

Student: Then what will be waiting for then?

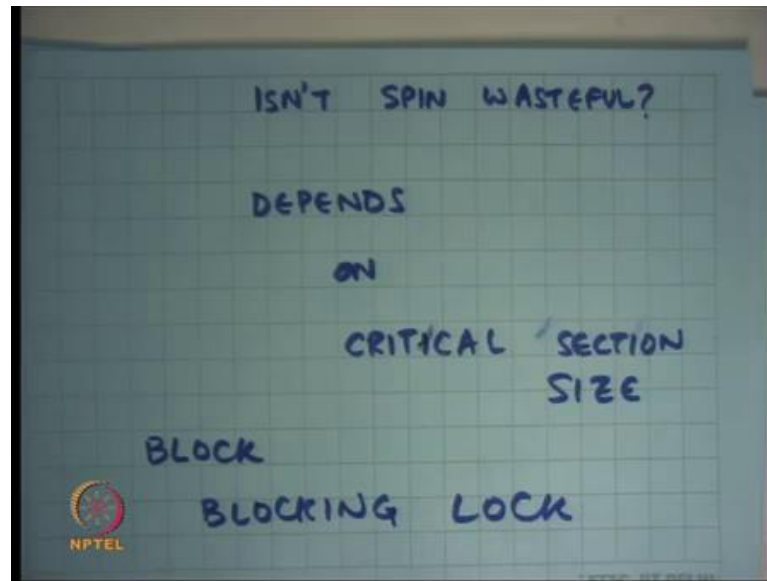
So good question the thread which is not able to acquire the lock we will just keep executing exchange operation again and again is not it wasteful dancer it depends and I am going to discuss, but let us first talk about correctness right.

So, good so we have checked if it was already locked. So, and this is correct right, because you have made sure that you are reading and writing this locked variable advance good. So, as I was pointed out one CPU will keep trying and other CPU will fall through. So, the one CPU that is trying is waiting and it is waiting in a loop all right. So, it is also called spinning all right. And, this type of a lock implementation is called a spin lock by the way how will you implement release.

Student: The previous (Refer Time: 37:52).

The previous one is right it does not matter just set it to 0 and that is fine ok. So, release a fine acquire only acquire need to change release was right. And, this is call, a spin lock ok. So, in this example what will happen if 2 threads come on acquire one of them is going to start spinning, the other thread is going to start running this code only when the first thread releases, the second thread which was spinning here is also going to get the lock and a second third is going to do it. So, basically serialized this operation of A and B and they basically doing it is in the exchange instruction right.

(Refer Slide Time 38:35)



So, now there is the question is not spin wasteful ok. So, what is the alternative? So, there is one answer which says can we disable interrupts would that be better just disable interrupts.

Student: And, then again enable it (Refer Time: 38:54).

And, then enable it at release. So, disable interrupts at acquire enable interrupts at release.

Student: Sir, not enabling and leave it once we have acquired the lock, we can enable the interrupts sir what is the problem?

Ok.

Student: Is the critical state is there (Refer Time: 39:09).

Ok.

Student: Acquired.

(Refer Slide Time 39:17)

```
struct lock {  
    int locked;  
};  
  
void acquire (struct lock *l) {  
cli → while (l->locked != 0) {  
    sti; cli; } atomic  
    l->locked = 1;  
sti →  
}  
  
void release (struct lock *l) {  
    l->locked = 0;  
}
```

All right. So, here is here is one suggestion. The suggestion is in this code here right enable disable interrupts here. So, it called prior cli clear interrupts and then re enable interrupts here right. So, on x 86 instruction is sti set interrupts right. So, call cli before and calls sti after and that should solve the problem.

Student: Sir, (Refer Time: 39:39) from the problem within the same CPU if we have multiple threads.

So, one answer is this will solve the problem only if there is a single processor. If, there are multiple processor it does not solve the problem perfectly fine all right ok, that is perfectly valid answer let us assume there was a single processor is this correct.

Student: Sir, it is some other correct, but (Refer Time: 40:01) exception then it will be (Refer Time: 40:02).

I have cleared the interrupts, I am spinning and let us say locked was equal to 0 and it was equal to 1. So, I am just waiting for it to get 0, if this is 1 part so, will it ever become 0, you know because I have cleared interrupts right timer interrupt cannot occur. And, so you know what is the hope? I am just basically deadlocked; it is not correct.

Secondly, you know a user program can definitely not do cli and sti right these are privileged operations. So, this privilege is only for the kernel, but even for the kernel this implementation is not correct.

What could have been correct? Well, if you really want to do it in this way one way to do it is inside this loop of while give the other thread of chance all right. So, just do sti and cli here all right. So, you are just disabling enabling disabling enabling interrupts. So, that there is another thread who gets a chance releases no deadlock right. This is actually a valid implementation of a lock on a uniprocessor; it is not a very very nice implementation right.

Student: (Refer Time: 41:12).

Student: Sir, this is also equally wasteful.

This is also equally wasteful all right ok. And, it does not work on a multiprocessor all right all right, but in either case in both cases we are doing spinning right, what is another, what is another way?

Student: Sir yield.

Yes good. So, the other way is yield. So, if one thread has figured out and it is not getting the lock you know as check the value of the locked variable it is 0, it is 1, it needs, it know that you have to wait why do you; why do you; why do you still going to going to CPU right. So, one way to do it is if we say that in that loop just call yield right. And, so, yield is going to suspend you and so, that way you may like other processes run, but does it really a better way.

Let us look at this example all right. In this case the critical section is very small it is just 2state 3 statements, which maybe you know maximum of 10 instructions all right. Or let us say maximum of you know 100 to 500 nanoseconds to execute these two statements right. The other thread how long would it have to spin, it will probably have to spin for around that time which is you know 100 to 500 nanoseconds.

How long would it take to actually call yield, let us say 1000 or 2000 instructions, which is much larger right. It is much better to just wait for 10 instructions rather than call yield which is going to actually change my process structures and all that and it is going to cause much better. So, in this case it actually better to use spinlock.

In fact, yielding is wasteful right was yield there is a cost to yield. Imagine if there were 2 threads that try to acquire the lock 1 thread got it, the other thread call yield. Now,

these threads will very soon release it right. And, the second threads should have actually now got the lock, but now it is in this separate code path it is calling yield and then later it will get scheduled after you know the next time interrupt or whatever and so, it is wasteful spinning would have been better.

On the other hand if the sizes of the critical section were large. Now, let us say you were executing 10000 instructions or if you were making some external access like a disk access right or some other slower device. If, you execute if your critical section is large in time, then it would be better to call yield, because for that amount of time you are just unnecessarily holding up the CPU all right. So, is not spin wasteful the answer depends and depends on what on.

Student: Size of critical.

Critical section size right. And, the basically the logic you want to use is if the critical section is likely to be large then use.

Student: Yield.

Yield right and the lock which uses yield is basically called a blocking lock. In fact, instead of using yield you could actually also use block right. So, what is block? Block basically just says that I want to block myself, because I know that this lock is not available. So, yield just says yield just puts you in the runnable state, which means on the next scheduling quantum you will again get to run.

Block basically tells operating system that I am not even in the runnable state right, I am in the blocked state, which means do not even schedule me and, at the release time the thread can say look at all the blocked threads and make them runnable right.

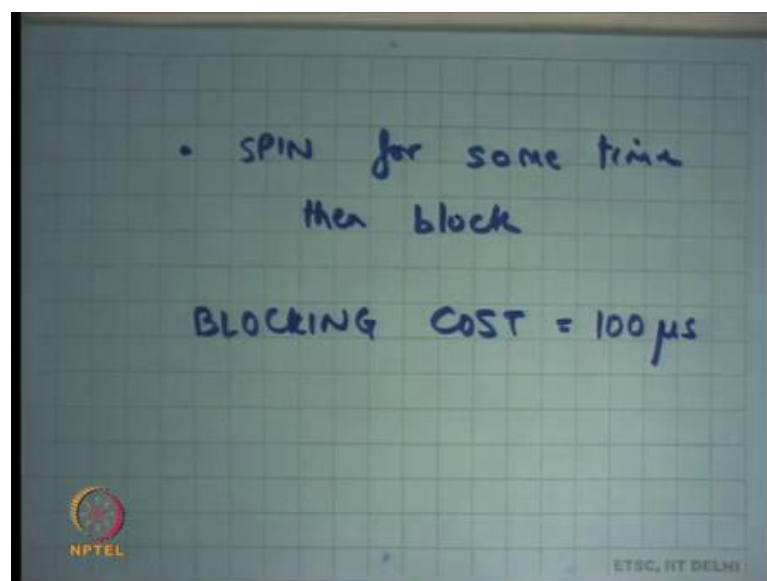
So, that is another way one is yield the other is block. Block is even more conservative; in the sense that if one thread is not able to get the lock it blocks, the thread which has the lock when it calls release is going to unblock all the block threads right. And, again there is some costs to block and there is some cost to unblock right and that kind of a lock is called a blocking lock right.

So, we saw a spinlock there is another lock called blocking lock, blocking lock will block the thread which call acquired and the release function becomes a little more complicated, because it unlocks the threads that have blocked on that lock right.

So, which one should you use? So, typically you will have both implementations available a spinlock and a blocking lock and depending. So, as a programmer you have to make a choice and you will basically look at your critical section size, typical critical section sizes and make a choice. It is possible that your code is such that some critical sections are very small, some critical sections are really large, and the same lock is protecting those critical sections possible right.

Some critical section is small and other critical section is large, the same lock is to protect needs to protect both those critical sections, what you do in that case? Well you can just say I will use a hybrid approach, I will spin for some time and then if I do not get a lock in that time then I will block right even that is a valid strategy.

(Refer Slide Time 46:38)



And, that is a strategy that is used in many lock implementations, which is spin for some time right and then block, while you were spinning you got successful you are done, but if you spin, if you spin for some time and you have not got the lock, it probably indicates that you know assuming that so, it is probably indicate that it is going to take a long time. So, it is better to now block right.

So, this is a sort of a hybrid approach adaptive approach where basically you at runtime figure out whether you need to spin or lock. So, how do you decide this sometime, whether it should be 10 nanoseconds, 10 milliseconds, 10 microseconds, 10 seconds? Once again it is a decision that you have to make depending on what is the cost of blocking right? How long does it take to block?

Let us say blocking cost is let say there is this variable there is this; there is this quantity called the blocking cost, which is how long will it take to actually block this thread. So, let us say the blocking cost is let us say 100 microseconds right.

So, if you know that the blocking cost is 100 microseconds roughly speaking, then 1 1 approaches that spin for 100 microseconds and if you do not get the lock in those 100 microseconds and block. So, now you want to so, the maximum amount of time you are going to waste is 100 microseconds of spinning plus 100 microseconds of blocking total of 200 microseconds right.

Student: Like you said yield is costly right.

Yield or block is costly yes.

Student: Sir, but suppose even if I am spinning we cannot proceed until somebody has actually released it, but for that thread to run actually it will have some time to it will take the time to switch off the angle.

Yes.

Student: (Refer Time: 48:51).

So, can you repeat I did not understand your question?

Student: Sir, I am that you said we do not in for small ones we do not need to call v.

Yes.

Student: But I am saying that if suppose another thread was running suppose one thread was got (Refer Time: 49:12) in the spinning state. So, we need another thread to actually release the lock right.

No. So, I am talking about multiprocessor system. On a multiprocessor system you have to make this choice between spinning and blocking on a uniprocessor system.

Student: We are always (Refer Time: 49:29).

What is the better choice?

Student: (Refer Time: 49:30) blocking (Refer Time: 49:32).

So, how many says spinning? 3 blocking all right most of the others yes. So, blocking is the better choice all right on a uniprocessor, because if you know if you are in a uniprocessor, you know you are trying to get a lock spinning is not going to solve the problem right. You on the you know uniprocessor, if you keep spinning on uniprocessor you are never going to see that lock released, because nobody else is getting the chance to run.

It is only when you get switched out will somebody gets a chance to run and he will release a lock. So, it is anyways the switching is going to happen. So, might it will just block right. So, when I am saying there is trade-off between spinning and blocking, I am really talking about a multiprocessor system, on a uniprocessor it is very clear always block.

Student: Sir, but user level even if it is spinning it gets preempt effect.

So, even in on a uniprocessor it is also to spin because you will get preempted. So, it is to spin, but from a performance standpoint it is better to block right. From correctness stand standpoint it is also to spin or even in a uniprocessor all right, ok, good all right. Let us stop here.