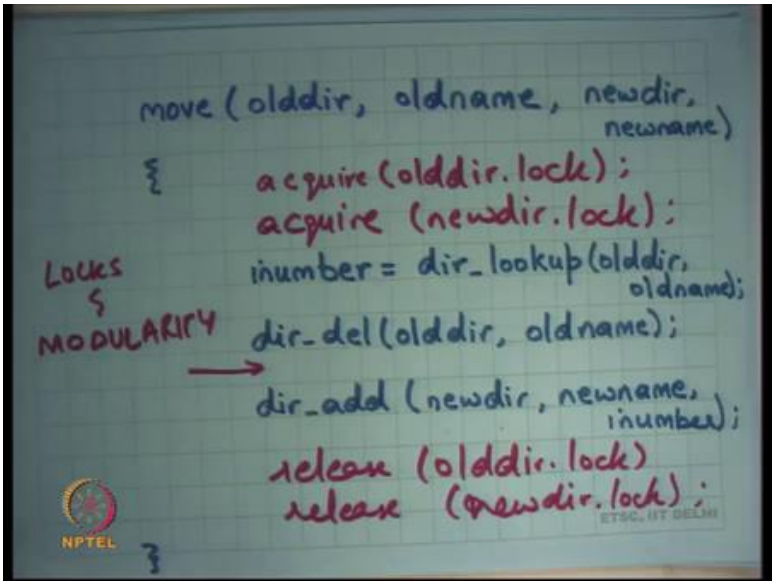
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 23
Locking variations

Welcome to Operating Systems lecture 23 right.

(Refer Slide Time: 00:30)



So, last time we were talking about locking, and we took this example this is hypothetical example of bank with many accounts, and with functionality like transfer, and sum and we said that look coarse grained locking can solve all the concurrency problems but coarse grained locking is not good, because it is serialize everything right. So, because it mutes in ensure that everything is mutually exclusive; it basically causes everything to get CU lies.

So, even if there are multiple processors only one transfer function will be able to execute anytime if you are using coarse grained walking. So, then we said ok, you know we should use fine grained locking and the question was how should you; how should you decide, how do use fine grained locking? So, this choice of how to choose where to use fine grained locking is a bit of an art. So, there is no, so I mean it is basically something that the programmer has to decide, based on what he feels is the right way of doing things right.

So, there is no one rule or to say that this is how you should fine grained lock in this program or that program, depending on the program you would not want to choose your fine grained locks differently. So, for example, you know yesterday we said that every account should have a lock so, there should be a per account lock. And any operation that require requires access to multiple accounts, you should take all the locks before doing that operations, all the locks for all the locks for all the accounts that are touched in that operation.

So, transfer operation touch 2 lock, 2 work rounds, you are going to take 2 locks, the sum operation touched all accounts you are going to take all locks right. And, we also said that one way to take the locks is to take it in an on-demand way. When, I say I take I take it in an on-demand way it does not mean that I released the previous locks right.

Because, this operation is basically an operation that needs to be atomic, we need to take all the locks at some point in time anyways, it is just that you can say that I know I will take the lock for the first count, then do some computation and then I will take the lock for the second count without releasing the count for lock for the first account and so on right.

So, you could do that, but we also saw that the locks have to be in a certain order to avoid deadlocks right. And so, the ordering the and the ordering has to be global. Once again you know the programmer has to figure out what the order has to be and the order will may be tied to your data structure, it may be tied to the semantics of your program. For example, the last time is we decided that we going to order it on the account ID of the account. And based on that we will take a priori all the locks needed for transfer we will take 2 locks, for some we will take all the locks, do our atomic operation then there is all right.

So, that was the that was a hypothetical example of course, let us look at another example, let us say I have a file system. So, as you know as an operating system one of the services that an operating system provides you is a file system. What is the file system? A file system is an on-disk data structure right.

So, a disk is nothing, but raw magnetic device which has lots of blocks and a file system is a data structure built on top of this sort of storage which allow and the semantics of the file system are usually the file system is hierarchical. So, you have a root directory and

then you have some names and each name may be a file or another directory and so on and so, you basically build a directory tree and that is basically what a file system is.

Now, you can imagine that there are multiple processes running in the system, multiple processes are making multiple system calls concurrently. So, one is calling read, another causing write, on different files, on same files all these are possibilities. So, question is the operating system needs to synchronize or make sure that operation accesses to the file system are correctly, you know correctly done and basically you know it basically means that operation should be atomic.

So, if there is an operation going on here and, in our operation, going on there, they should not appear interleaved at any point, because interleaving of those operations can cause bad things in your file system all right. So, you know one option is once again coarse-grained locking put a lock on the entire file system; you are safe, definitely safe right. But of course, that is not a very good solution you can imagine that your system will run at very very slow speed. Now, nobody will be able to access the file system concurrently only one person will be able to access the file system at the end.

So, what do you do? Once again choosing what locks to take is a bit of an art you may say let us have a lock for a directory, or you may say let us have a lock for file, or you may say let us have a lock for you know just very hypothetically. Let us have a lock per pair of files, you know if you figure out that most of the operations are actually occurring on pair of files.

So, why not you know have a lock in sensation per pair of files and if you are going to you know and do an operation between those 2 files or something but you know when in that case if you are going to touch one file then you would take all the locks in which for that file where that file belongs to a pair.

So, if you know for all the pair for that file you need to take a log so, that does not make a lot of sense. So, yes, I mean you know intuitively it seems like the best thing to do is basically take a file for lock a lock for file all right. So, some what I am going to show you is basically you know if you do this kind of fine-grained locking it hurts your program structure. So, if the program structure does not the modularity in your program actually reduces because of this.

Because of the locking behavior. So, let us say because of fine grained locking basically. So, let us say I have a function which looks like this, it says move so, just moving a file from one directory to another directory.

So, it says move this file name called old name from old directory and put it as new name in new directory right. So, that is the semantics of this function and what it does is it basically looks up looks up the disk block. So, let us say i number is the disk block or some identifier which is identifying the number at which this file is stored just looks up the old name in old directory, delete deletes old name from old directory and adds new name to new directory, that i number that you looked up right.

And, so this code is correct let us say when you run running serially. When there is only one thread that is accessing it, this code is also correct if you are having one big global lock that is protecting this entire function but let us say I have per file locks right. So, or per directly locks, so, let us say I have per directly locks. And, so what do I need to do I am accessing I am accessing the old directory, reading, and writing the old directory here. So, I need to have I need to lock this region with old directories lock, and I am adding something to new directory.

So, I need to lock this region with the new directory locked, but can I do these in isolation? Well no because you know I want perhaps I want my move operation to be atomic right, if I just say that over let you know that let directly delete do the locking inside it and I do not care about you know what locking it does inside. And, then let direct directly add do the locking inside it and I do not care then what happens is at this point here no locks are held and anybody is free to observe these directories or the state of these directories and at this point what you are going to find is that this file does not exist anywhere right.

And, so this the file system is in an inconsistent state at this point, you know so there is some there are some disk blocks that do they are not pointed to by anybody, neither by the old directory now nor by the new directory and that is an inconsistent state right.

In other words, you know if you do it in that way the move operation is not atomic right. So, what you what would you want you would again want to do basically something like this, you would say acquire old dir dot lock and acquire new dir dot lock right. And, then you will do this operation and then you will just release these locks.

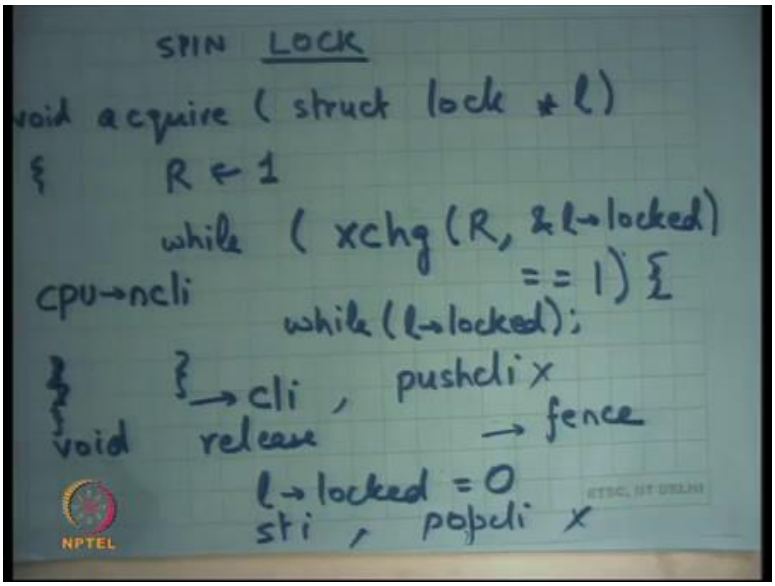
So, what has happened is basically because of fine grained locking any function that is building upon these so, earlier it was very modular, you know move function could have been written in 3 lines and without having to worry about what these functions are doing inside. Whether these functions have to take a lock, does not take a lock that is not my business, I just call these functions. But now because I am doing fine grained locking now it is my business to know, what locks are they going to take right.

And in fact, instead of asking them to take them I need to take them on their behalf, and I will need to take them in a certain order right. So, in other words basically what I am saying is locks and modularity are sort of you know so, locks basically hamper modularity, it is a locks are not very friendly to modularity they sort of make your code more complex less modular right.

Earlier you could just say that this function is going to do delete, this function is going to do add, I do not care what it does internally, but now you have to worry about this function is actually going to need to take a lock, and this function is going to need to take a lock, and because I need to do this atomically instead of them taking a lock let me take a lock on behalf of them. And, now because I am taking a lock, they should not be taking a look and so on right.

So, the entire semantics of your function has become complicated. These semantics are not just that this function this function is going to delete it a name from the directory, the fine semantics now need to be this function is going to delete a name from the directory, and it should not it should assume that a lock has already been taken and it should not be taking a lock it itself right. So, I mean locking and fine-grained locking especially sort of complicates things right. So, let us look at locks and locks implementations in a little more depth alright.

(Refer Slide Time: 10:28)



So, we will we said that our locks implemented you know one of those one of the; one of the ways we implemented locks in couple of lectures back was a spin lock, and where we said that there is a function called acquire right, and the let us take struct lock star l let us say and it just says while and let us say this is a function which is internally calling the exchange instruction.

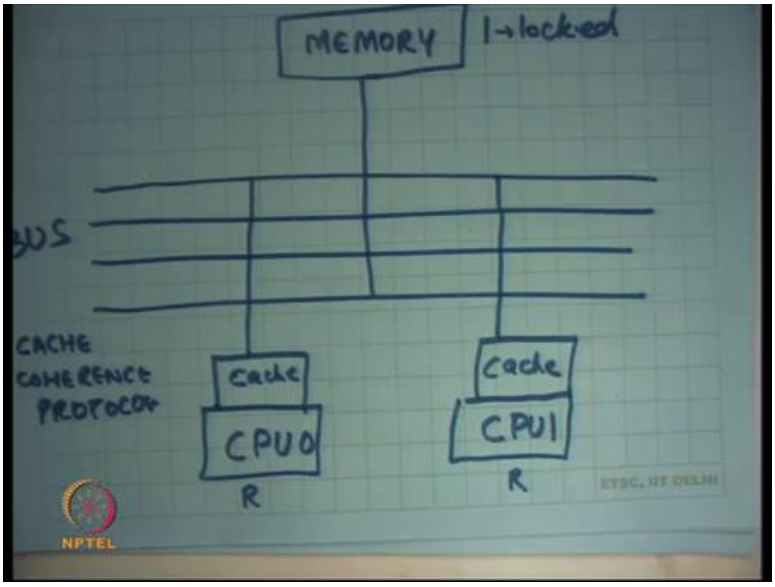
So, if I am calling the exchange instruction. And, so you know one way to do this is let us say there is a register, which I have put a value 1 into and then I say while exchange register address of the locked field in l is equal to 1 I keep spinning otherwise I (Refer Time: 11:19) right. So, you know just read this code once more, basically what I am doing is I am basically trying to put the value 1 into the locked field of this l right. So, I basically want, I want to put a value 1 into the locked field of the l variable except that I want to make sure that earlier it was 0.

Then if it was early as 1 then I should be just waiting for it to become 0 right. So, that is basically acquire that is the semantics acquire and this how I am implementing it and we seen it before. So, I put the 1 value in R and this function is going to atomically swap R and this memory location l dot locked right. And, so if l dot locked was 0, R is going to become 0 and so, you going to come out of the loop, but if l dot locked was 1 then R is going to remain 1 and you are going to retry, but making it you know retry it till you see a 0 value and locked right.

And, we also talked about last time why this implementation is an atomic or you know it works because if 2 instruction 2 threads tried to call exchange simultaneously one of them will occur before the other, they cannot get intently. So, this swapping operation is that all basically right. So, everybody remembers this right all right.

So, let us see what happens at the hardware level then you execute something like this alright. And just for completeness let me just also write it release; release is just l dot lock is equal to 0 ok all right. So, let us see what is happening at the hardware level.

(Refer Slide Time: 13:02)



So, let us say here is my bus right. So, we have seen this diagram before, I basically always draw a bus here and I say that here is my CPU right. And, let us say the CPU 0 and this is CPU 1 right and let us say this is memory ok. And, inside the memory there is this variable called l dot lock.

Student: (Refer Time: 13:37)

Right. And, in the CPU 0 there are private registers Rs right. And, what I am what each let us say both the threads are executing simultaneously on CPU 0 and CPU 1, this thread is going to set it to 1, this set is going to set it to 1 both are going to say exchange one of them is going to win, whoever wins gets a lock the other one just spins, that is what is the idea. Typically, you have must have studied in your operating system class or in a computer operating class that every CPU also has a cache right. So, let me just say cache.

So, my first question is when I call the exchange instruction is it to just exchange from within the cache? So, l dot locked is just another memory location right and so, when you access it, it just comes into the cache. And, can they exchange the instruction just you know do the local operation without having to go down under disk on the bus?

Student: No.

No because you know because l, because the exchange operation is an atomic operation and they need and there needs to be serialization between who is doing this you know. So, there has to be some communication on the bus either the communication has to be directly with the main memory, or they have to talk with each other. To basically make sure that you know there is serialization either heavens or heavens right.

So, one of them is going to get 0, another one is going to get the answer 1, both of them cannot get the answer 0 basically. And, so there has to be some bus protocol here that has to happen here and so, each exchange instruction will require some bus transaction right.

In general memory accesses do not necessarily require bus transaction right, whenever I read right a value if the value is found in the cache, I can just locally satisfy it from the cache. It is only when there is a cache miss, I need to go to the memory right. And, typically you know these processors has what is called a cache coherence protocol.

So, the idea is that let us say I access the memory location a and it gets cached here and then this CPU accesses the memory location a then you know there is some protocol that is going on here which will invalidate this location and then valid it and then bring it here right. So, you know if these both the CPUs are accessing the same location then, there will be some bus transactions that are shuttling this variable between these two right.

In any case you know when we are doing this exchange business then the problem is that it is there is a lot of bus traffic basically going on. You know if there are 2 CPUs there is a certain amount of bus traffic, if there are 4 CPUs there both there is more their 8 CPUs even more if the 64 then you know basically bus is definitely the bottleneck. So, cache coherence protocol is for every memory access all right.

So, for every memory access clearly, I mean you cannot have so, the hardware ensures that you know there is some sort of so, there is that is what coherence means. So, there is

coherence in accesses, it cannot be that the same location has two values basically at the same time. So, for every memory access the cache coherence protocol works. It need not works or if the same CPU access at the same location 10 times, it is only the first time that there will be a bus transaction.

The next 9 times it will get satisfied from the local cache, without any bus transaction, without any cache coherence protocol getting having to kick in because assuming that this other CPU is not accessing that location that location is locally satisfied from the cache all right. But if you are executing the exchange instruction each time then you have to make a bus transaction because it has to be atomic with respect to everything else right.

So, in our in the code here; let us say if there are you know if there are; if there are 4 2 processors; one of the processors gets the lock, the other processor just keeps calling exchange and the exchange all each exchange execution exchange is causing a bus transaction and so, there is a lot of bus traffic ok. So, this is not the best possible implementation of a spin lock. And how can you make it better well one way to make it better is for example, put another loop here which is not using an atomic exchange operation, which is just checking.

So, exchange of instruction is a more cost is costly operation because you know it needs atomicity. On the other hand, this operation is just a read operation; a read operation is a less costly operation it does not need an atomicity right. And, so what I am doing is I am basically you know instead of every time calling the expensive operation, where I am basically doing is I am I want to wait; I want to wait for lock to become 0 right. And, so instead of doing it in, but I am doing I also need the exchange interaction, because I want to sort of swap it atomically.

So, that checking code can be done through just reading and then once you are, once the read says yes it has become 0, then you can retry the exchange operation. It is not necessary the exchange operation will succeed, but it is a high likelihood that will succeed this time right. If it does not succeed no problem, you again come back here, and you again wait for it to become 0 right. So, what will happen in this case?

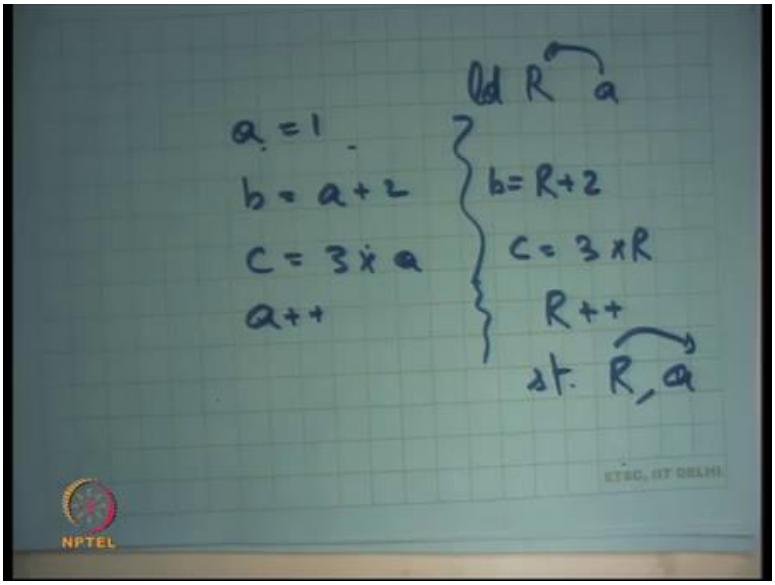
Let us say you know both CPUs try to do exchange one of them wins, the other one just calls the loop and this time that will the inner loop is going to get satisfied from the

cache right. So, the inner loop is going to get transferred from the cache you have reduced the bus traffic all right all right.

So, this is all good, but let us see what happens, if you write code like this you know without having to you know let us say you write this code in C, you just say while exchange and then in the inner loop is while locked. You know if you know a compiler is basically looks at these variables and decides which of these variables to register allocate and which of these variables to keep in memory right.

So, what happens if this variable becomes register allocated right? So, I hope people understand what is register allocation of a variable studied in the programming languages class or right?

(Refer Slide Time: 20:05)



So, basically the idea is let us say; let us say there is a variable called a, and say a = 1, you know b = a + 2, and c = 2 3 into a or whatever. And, so the question is one way to deal with a is basically say that keep it in memory and each of these operations are memory accesses. And, there is other way is basically read a into a register.

So, let us say this is a load instruction and you read a into a register and then you perform all these operations on R. So, you say you know R + 2 = b and c = 3 * R and let us say you also say a++. So, you say R++ and then later on you can say store R to a right.

So, this is a common optimization a very most basic optimization of a compiler, that if there is a memory it is as a variable instead of so, variables are basically you know have a one to one relation with the memory location, but if there are multiple access to a variable, and the program and the compiler can see that there multiple access to the variable. The optimization is that you just bring the variable from memory into a register do those accesses to the register instead of the memory so, you have saved some memory accesses.

And, then after you have computed at the thing you just save it back into the memory right, common optimization of a compiler. Similarly, in this code this variable l.lock a compiler is free to register allocate. So, what can happen if the variable gets register allocated?

Student: (Refer Time: 21:38).

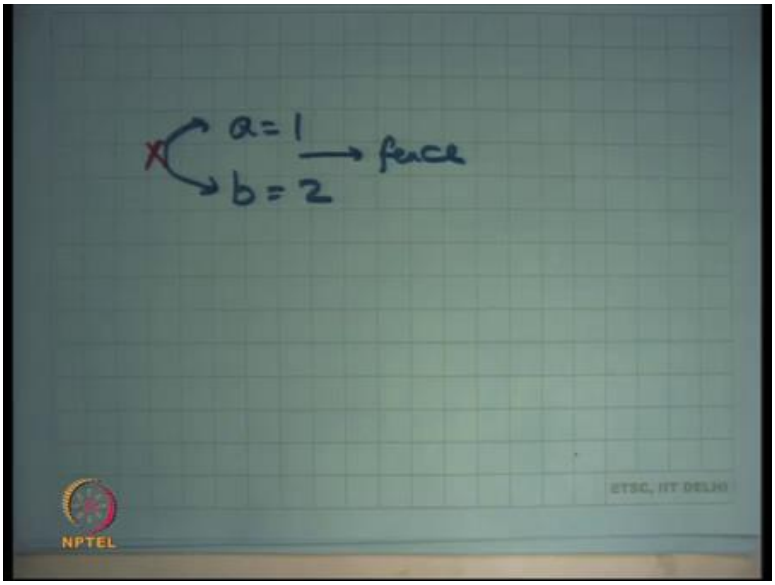
It is an infinite loop right it will never finish. So, you know with the best of the intention compilers are not really you know playing well with what the operating system designer really wants. And, so you know either the operating system designer writes this loop in assembly or actually the compilers give you special keywords to basically say do not optimize this variable all right.

So, there is a quick for example, and see there is a variable called there is a keyword called volatile. So, if you declare a variable or a you know field with a volatile struct or with the volatile type. Then basically the compiler says this is something that you know the programmer has really written very carefully I should not be optimizing it at all right. So, you know just an interesting example of how you know, how a compiler writer. So, a compiler writer does not worry about concurrency and does not need does not understand which one what is the lock and what is not a lock etcetera.

He is just looking at code and he is just you know optimizing it, but you know if you are writing the special code like this you should basically declare things as volatile. And, this is one of the reasons why you know it is right difficult to get concurrent programs correct. Notice that you know it is easy to basically say that acquire right this acquire function very carefully.

And, then use this acquire function to mark critical sections, but on the other hand if I did not want to use locks and I just wanted to very carefully write this sort of code then I have to worry about over the compiler should not optimize it and you know other things like that. And, so that is a very hard thing to reason about in general all right. The other thing a compiler and even in the hardware does is reordering right.

(Refer Slide Time: 23:21)



So, if I basically have an instruction it says a is equal to 1, and then I say b is equal to 2; a compiler is free to reorder these instructions right. For a compiler these are completely different memory accesses completely different variables, it does not matter which occurs first right.

On the other hand if you look at our locking code you know reordering is fatal for our logic, you know because if we are writing to the lot field and then we are accessing some shared variable, if the compiler reorder these things, then you know the critical section is outside the lock or before the lock and bad things can happen right. So, similarly it is possible that the release the critic an access in the critical section is reordered after the release right.

So, for example, in this case l dot locked is equal to 0, before that is I had a shared variable access you know these two come are completely independent memory accesses and you know a compiler may say let us just reorder these things. It is not just the compiler who can do this; it is actually even the hardware that can do this right. So, most

so modern hardware basically does out of order memory accesses right even the Intel architecture and most of the performance they get are basically because of out of memory accesses.

And the reason you need to do out of auto memory access is because some memory access is going to take a long time, and others are going to take a short time, because some memory accesses maybe cache hits and others maybe cache misses. So, if whatever the cache hit you know let us just do that first and what does the cache miss do not let it come whenever it when it is ready right. So, even the hard even if the compiler played well with you the hardware can actually reorder accesses.

So, it is possible that the locked access locked variable was in cache and just sort of got you know get what said first and later on the other critical sheared variables getting set. So, once again you have to be very careful in doing this and so, you know modern processors provide what are called fences right.

So, we basically put a fence and the fence is basically saying that all memory accesses before the fence should have finished before any memory access of the after the fence starts ok. So, the idea you know from a hardware designer stand point is that in general let us allow the ordering of memory accesses, reordering of unrelated memory accesses of course, which seem unrelated at least, but a programmer has a way of saying that here is a memory access and here is the memory access.

So, in this case if I want to disallow this so let us say I want to say that this is this should not be possible, then I will put a fence in the middle, so, that is you know. So, there are multiple ways of putting a fence it is very architecture specific you know you have special instructions which you can say that you know his fence there is a fence instruction.

So, you can put a fence instruction. So, that way this will get disallowed or there are special instructions like the exchange instruction it itself acts as a fence right. So, some instructions will never allow reordering of across them themselves all right good no all right.

So, let us look at this implementation again. So, what I am saying is that the exchange instruction itself is acting as a fence in the case of acquire. And, in the case of release the

programmer should put a fence in some way or the other right either a fence instruction or instead of using a simple right you will use some exchange instruction to do the right for example, all right ok.

Now, let us say, so this is a spin lock right and let us say I am an operating system developer and I basically also get interrupts. So, these spin locks will protect against multiple concurrent access by multiple CPUs, but if I am within let us say I am within in within the critical section and an interrupt comes the interrupt handler will get to run. And, if the interrupt handler also needs the same lock then there are problems right, you can either end up with a deadlock or.

So, if you are doing it like this then if I am within a critical section, and I am holding a lock, and it is possible that the interrupt handler also wants to get the same lock then, I will have a deadlock right. So, and you know the most the core of the operating system typically has such code. For example, there is a lock to protect the process table p table in xv6 for example.

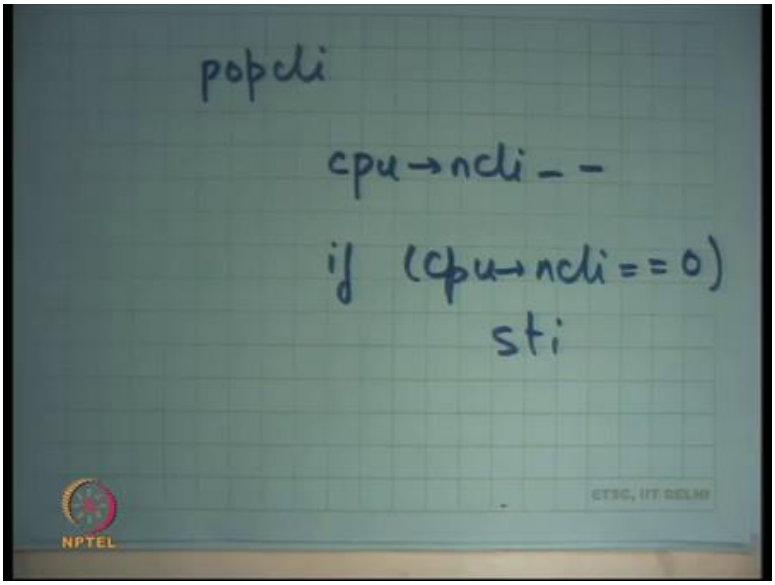
So, that lock is you know is being accessed by multiple functions and even the interrupt handler the timer interrupt handler is going to need to access this p table lock right. And, it is going to need to access the p table and we are going to need to acquire the p table lock. So, such locks are you know or even more special. And, so what you do is in that case you basically make sure that not only do you just do this you also disable interrupts in your acquire and you are enabling interrupts in release right.

So, you know when you acquire a lock any lock, if you know that these locks can be acquired by or can be requested by interrupt handlers you also disable the interrupts. So, within the critical section and interrupts is not possible anymore right. It is only when you really you quit the secretarial section will an interrupt get in the way and. So, you know one xv6 you will find a function called instead of just cli and sti you will find push cli, and instead of sti you will find popcli, and the idea here is that it is possible that you know you are trying to acquire multiple locks.

So, let us say you acquire p table lock first and then you acquire file system locks second and both of them wanted to you know both of them need to do cli, but then let us say you release one of those locks then you do not want to immediately do sti. So, basically you

have some kind of recursion, so each CPU. So, there is a CPU pointer dot so, there is a there is a CPU.ncli variable.

(Refer Slide Time: 29:28)



And, so push cli just does cpu.ncli++ and if cpuncli = 1, then you actually call cli right. Which means you just transition from 0 to 1, so, you actually need to disable interrupts and similarly pop cli.

So, that is push cli roughly speaking and that is pop cli. So, pop cli is basically cpu pointer on ncli-- and if cpu.ncli = 0 then sti right right. So, clearly, I am talking about within the operating system, where the interrupt handler can require would need to acquire the same lock that you are holding right, only in that case do you need to disable interrupts. And, I am really talking about that the real inner core of the kernel ok.

So, for example, then you see implementation of the spin lock in the xv 6 kernel, and the spin lock is basically used for your p table lock, and among other things you would basically find that the acquire function not just does the exchange to protect against other cpus it also does a cli to protect against interrupt handlers ok. So, I need to do both these things alright. So, and also this ncli variable is a cpu private variable all right. So, there are ways to say that this variable is only going to be accessed by this cpu.

And, so no other cpu can will ever be able to access that value, or you can just have an array with you know first where each element is accessed by only the corresponding cpu and nobody else so, that is a portal per cpu variable.

(Refer Slide Time: 31:17)

So, let us say I am in the user mode ok. So, I have talked about kernel mode, but let us say I am there is a user mode. and I want to do I want to implement my let us a banking application and I want implement locks. So, what kind of locks should I use? Well firstly, question could be you know whether I am running on a multiprocessor or a uniprocessor? Or in fact, even before that the question should be whether you want to implement a spin lock or a blocking lock, right?

So, if you want to do a spin lock do you need any kernel involvement unless you want to disable interrupts? What a user level lock needs to disable interrupts? I said you need to disable interrupts only if that lock could be requested by an interrupt handler. I mean assuming that the user level locks are just private to the user and the kernel has nothing to do with it, then the interrupt handler has nothing to do with that lock right. So, you will not need to disable any interrupts for a user level lock ok. So, can you do implement a spin lock without having any kernel involvement? The answer is yes ok.

All you need to do is declare a variable and use the exchange instruction; exchange instruction is an unprivileged instruction right; it just has the semantics that things will be atomic that is all right. So, the same code that I showed you this one without the cli

implements a spin lock in user mode ok. So, a spin lock in user mode is as fast as a spin lock in kernel mode you just basically you know try to atomically set it to 1, and if not you just spin just in exactly in the same way, and hopefully your critical section was small and you will immediately get the lock all right.

Does it matter whether you are using kernel level threads or user level threads, because user level threads will only run on a single CPU you do not even need to do this exchange business right. User level threads will only run on a single CPU and so, instead of using a spin lock you would probably want to use a blocking lock instead right.

And of course, so, blocking locks will be used either, if you are using user level threads or if you are sure that you know your threads are not going to run on a single CPU for whatever other reason there could be and if your critical sections are known to be very large right. For example, if you are making a system call while holding that lock you might as well just you know use a blocking lock, rather than using a spin lock right.

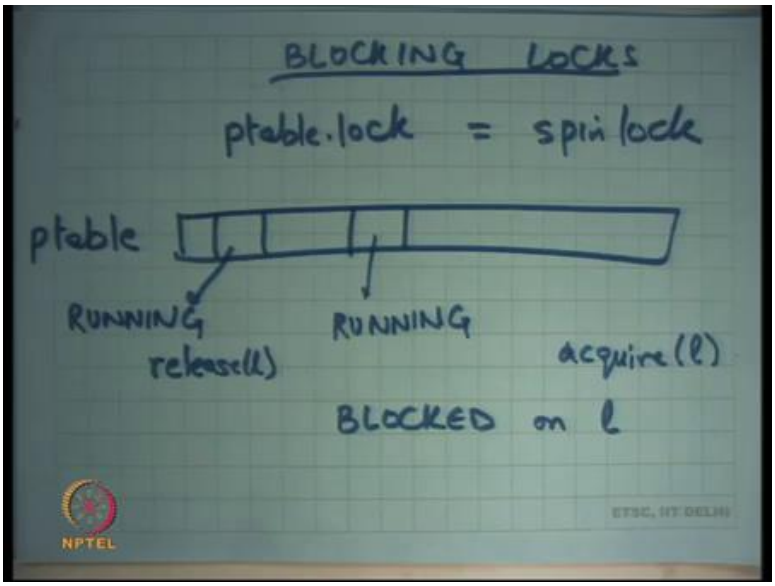
So, in all these cases you will not even spin lock you will use a blocking lock. Do you need kernel involvement to do blocking locks to implement blocking locks?

Student: Yes.

Yes, because a blocking lock basically needs to tell the kernel to change my state from ready or running to blocked right. And, so I need and the only the user has no way of changing it from ready to block and so, it has to tell the kernel to do it right. So, there has to be a kernel interaction unless of course, you were using user level threads in which case the kernel has no idea and so, you are in that case your user level scheduler is just going to is just changing the state of your you know a currently running thread to from ready to block.

So, in that case you are p table is maintained at the user level. So, in either case the p tables, the state in the p table needs to be changed from ready to be blocked. If you are running kernel level threads you need kernel interaction to do that if you are running user level threads you can just do that locally in the user all right. So, let us see how blocking locks are implemented right.

(Refer Slide Time: 34:48)



So, you can imagine that there is a p table right or you know I am using a an array, but you could even have a list of PCBs, or any such data structure that is maintaining all your process PCBs, Process Control Blocks right. And, what you are going to do is let us say somebody says lock and he is not able to get the lock then you will basically want to change its state.

So, let us say this is a process and this is currently running, then and it calls acquire you would want to change it is state to blocked right. And, you will want to record that it is blocked on whatever was argument of l acquire, so let us say blocked on l right.

And, then if somebody calls some other process so, this becomes blocked. So, this never gets to run in future till somebody calls release. So, let us say here is the process that was running and then it calls release l. And, what release is going to do is it is going to go over the p table right and pick up one process, that is blocked on l right. So, this from here this here the ls are matched and change it from block to running already not running, but ready it will change it from block to ready all right. So, that is how blocking locks will be implemented all right.

So, but this p table structure itself it needs to be protected, you know accesses to the p table structure it says by multiple threads needs to be protected. So, you will use a spin lock to protect the p table and then and so, blocking lock internally will use a spin lock to protect this structure and to switch from running between running and blocked these

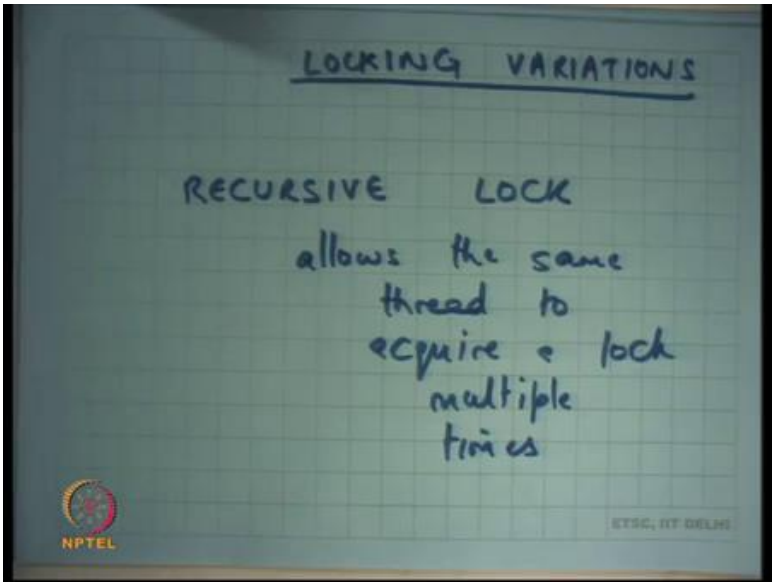
different entries right. So, there will be as p table dot lock let us say which will be a spin lock.

Student: P table dop lock (Refer Time: 37:06).

Well I mean will the p table dot lock only be needed for a multiprocessor, well you know. So, on a uniprocessor a p table dot lock equates to a cli no clear interrupts. So, basically you want that while you are in the middle of accessing the p table nobody else should basically interrupt you right.

So, on a multiprocessor you will use a spin lock on a uniprocessor you could do that just by disabling interrupts all right. Basically, what you want is mutual exclusion why you are accessing the p table right. And, mutual exclusion on multiprocessor only base spin locks mutual exclusion on a uniprocessor the way is disabling interrupts.

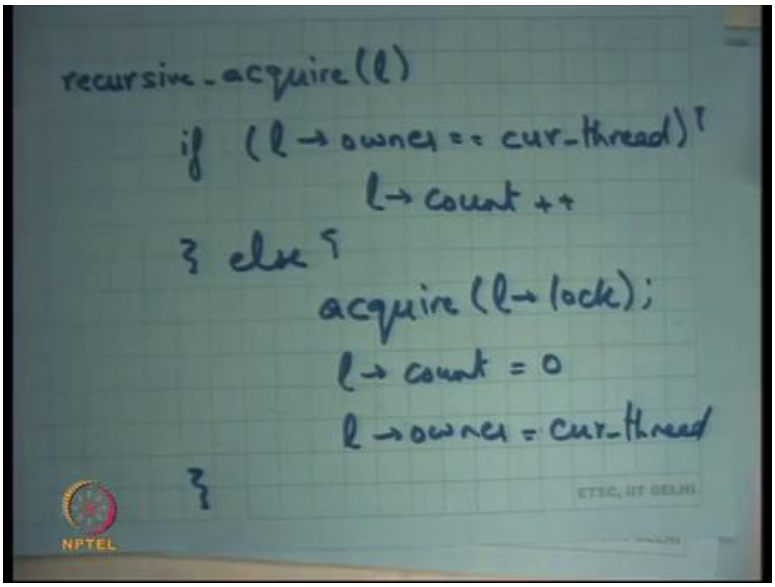
(Refer Slide Time: 37:49)



Now, let me talk about some locking variations all right. So, there is something called a recursive lock. So, you may have seen that sometimes we run into this situation where, you acquire a lock and then you call some other function and that once you acquire the same lock, and at that point we deadlock right, because the same thread cannot acquire the same lock multiple times.

So, you know the recursive lock basically what it does it allows the same thread to acquire a lock multiple times all right and the semantics of a recursive lock up fairly simple.

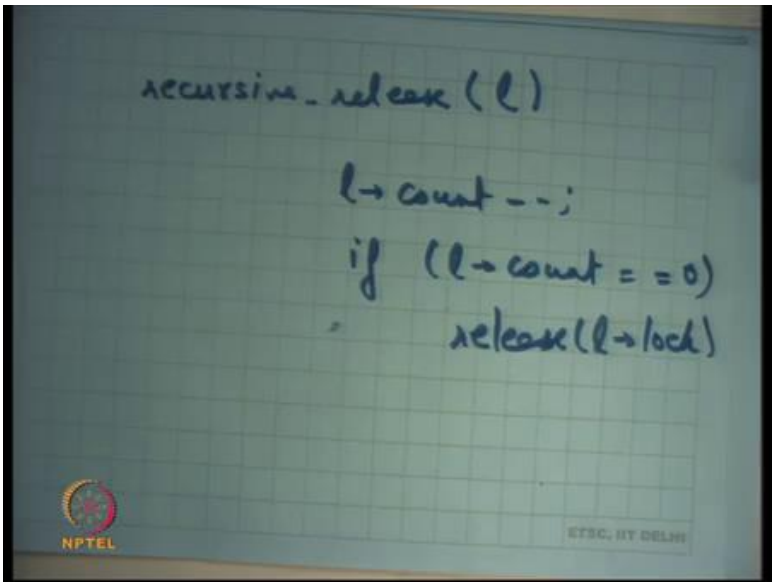
(Refer Slide Time: 38:52)



Let us say this is recursive lock; let us say this is recursive acquire l, you will say if l.owner you will keep something called an owner is equal to current thread, then l.count++ all right else you call the regular acquire all right. And, you set l.count to 0 and l.owner to cur thread right.

So, basically the idea is you know a lock is supposed to provide mutual exclusion between multiple threads. If for some reason the programmer feels that you know or for modularity or whatever reason, if he feels that the same thread wants to acquire the same lock multiple times let us allow that right. So, that is a that is the; that is the spirit behind a recursive lock. And, of course, release will just basically decrement count and only if count becomes 0 does it release the lock right. So, that is what relieves me does. So, should I write release?

(Refer Slide Time: 40:13)



So, let us say let me also write release, I will just say l.count-- if l.count = 0, then release l.lock right something like this right. So, this is a recursive lock sounds like a good idea or a bad idea? No idea ok.

Student: [laughter].

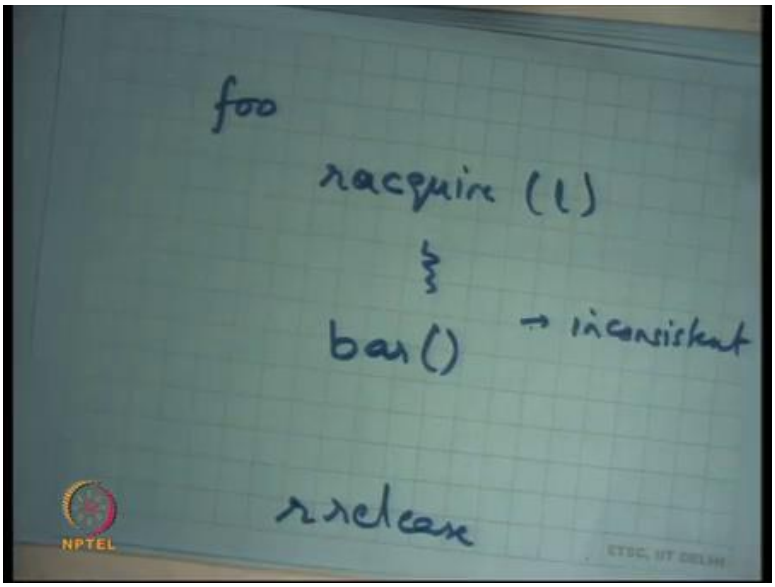
So, it is actually a bad it is generally considered a bad idea to do a recursive lock all right and why. Basically, usually the semantics of a lock is that when you acquire a lock you know at the point, when you require the lock and you just enter the critical section you can pretty much assume that this that the state is it there is a consistent state of the system right.

So, if the idea is that if you have been able to acquire the lock, anybody else who has released the lock has left this has left the state in the sheared state in a consistent state right, where is less that left the memory in a consistent state right. That is basically; that is basically have that is basically been are invariant right that if I am able to acquire the lock I can assume that at the first instruction of my critical section, the system is in a consistent state.

And that other invariant I usually maintain is that just before I release the lock, I have ensured that the system is again in consistent state right. And, so then I release the lock, so, that the other person who acquires it will also see the system in a consistent rate. So,

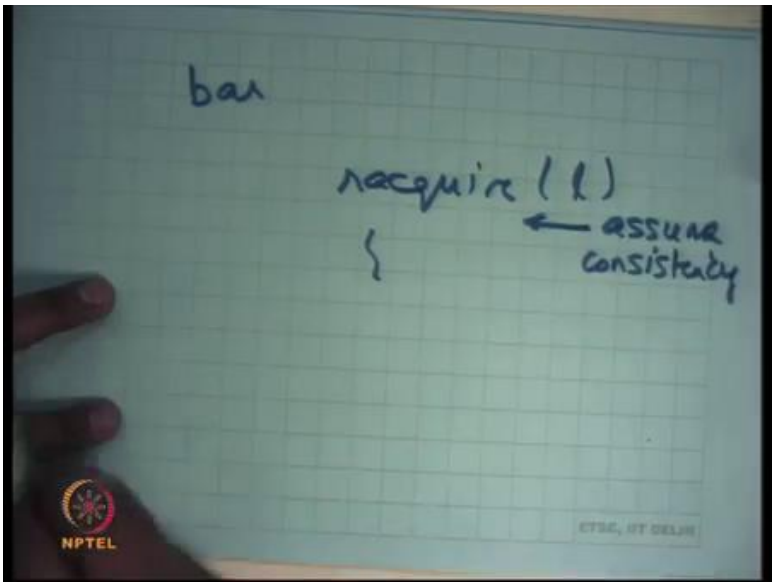
generally you know you the assumption is that as you have acquired if you have acquired the lock the system is one consistent strait and you will maintain it in a consistent state before you release the lock or you will keep it in the system.

(Refer Slide Time: 42:15)



But, if you do recursive acquire then you know then it is possible that you have a function foo that you know says let us say I am going to say recursive acquires r acquire l, does something makes it inconsistent right make the state in inconsistent has not released the lock right yet. So, he is going to say r release here somewhere here, but he calls bar here right.

(Refer Slide Time: 42:37)

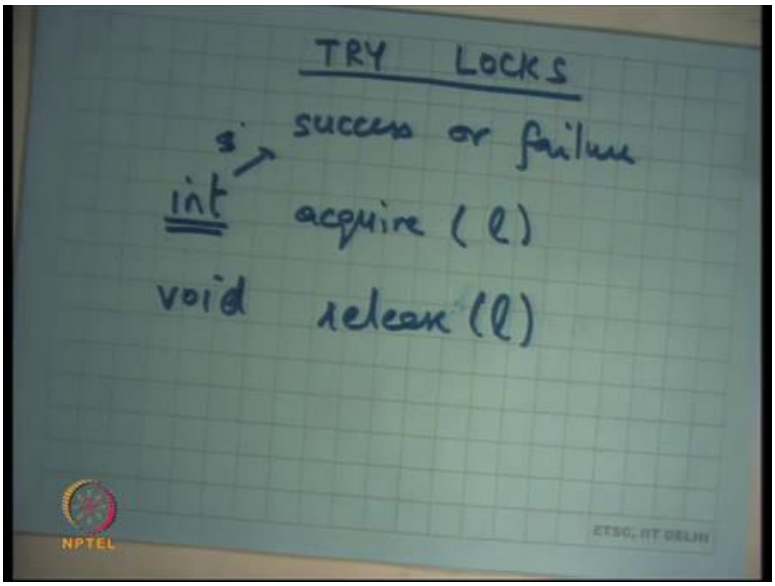


And, bar internally is going to say r acquire and he is going to start doing something, but he is going to assume you know assuming that these code has been written you know in modern modular fashion in a different file or different program or whatever he is going to probably assume that it is inconsistent state, for assumed consistency. But because you know you are using recursive locks you will you know you will violate that assumption and this bug will be much harder to find.

So, in fact, you know using recursive lock, you have made it easy for your program to have bugs right, that have not been that cannot be found. On the other hand if you did not use the recursive lock, you know the first call to bar would have told you over there is a bug in a program right, because there would have been a deadlock right there right.

So, in general you know a programmer wants to keep his thinking simple and consistent with this idea that when you get a lock, things are consistent when you release the lock things are consistent. And, if the programmer is indeed doing that then recursive acquire is a bad idea all right ok.

(Refer Slide Time: 43:44)



Then there is another variation of locks called try locks ok. So, what are try locks? Instead of so, the it is the same thing let us say instead that so, the idea is that acquire l has a return value now int right, which basically says and you know the release is just void and the acquire basically says success or failure right.

So, in our regular lock and acquire basically always succeeds or it waits. In the case of a try lock you try to get the lock. If you did not get it you just return a minus 1 or you know a failure value right and when you and so, it is up to the caller to do whatever he likes of course, you know you can implement a regular lock using a try lock very easy.

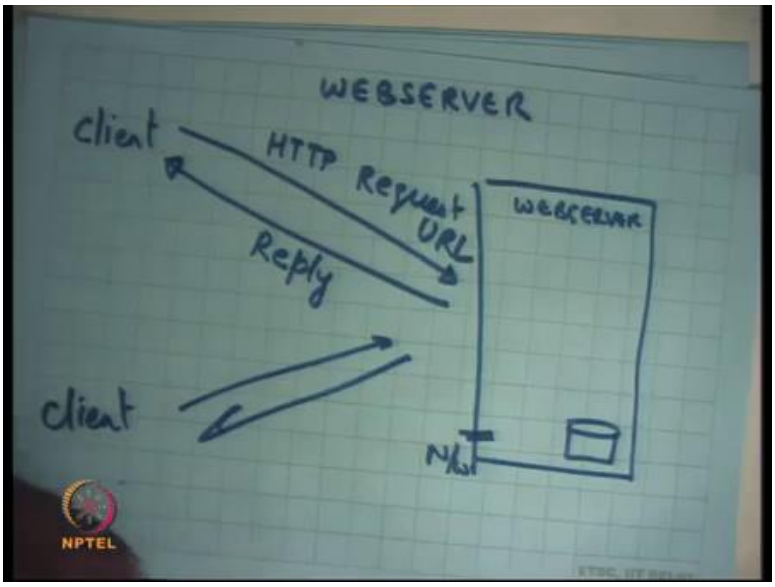
You can just sort of put the try lock in a loop and you get a you get a regular lock; it may not be the most efficient way to do a regular lock right. But the advantage of a try lock is that gives some flexibility to the caller, he may want to say let us try to acquire this lock, if I do not get it then I have something else to do let us do that first right and then retry it. So, it gives him that flexibility.

On the other hand, and the previous lock and acquire basically is committing that I am definitely going to I am going to either do that or wait basically right. So, try lock gives you some flexibility into you know whether you want to whether you want to wait or whether you want to and do something else.

Let me now discuss a real example. So, I hope you all know that the banking example that I took earlier was a very very hypothetical example for many reasons. Firstly, you know bank accounts are not maintained in memories, secondly, usually do not write code in such a way where you are going to do a global sum operation on all the accounts, you would want to do some kind of more distributed and segmented way of calculating sum.

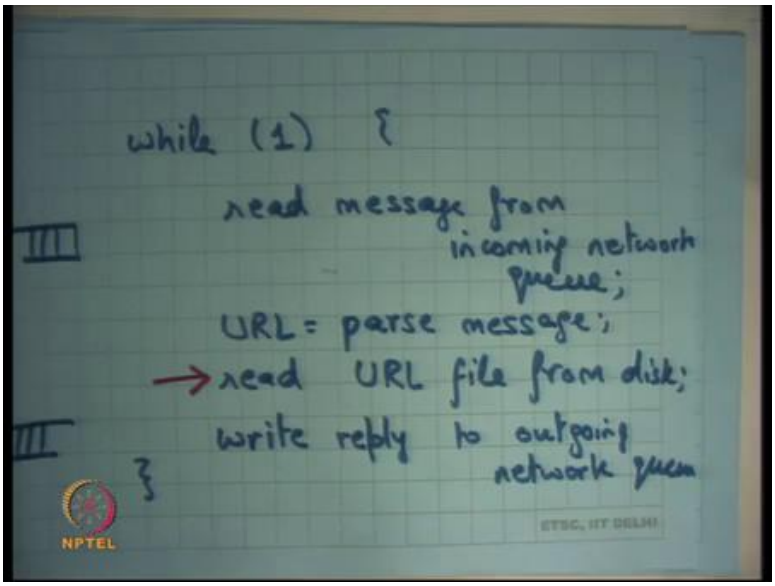
And, so that there is more scalability in your system or you know whether you want to calculate some at all you can just you know update the sum as the transfer is going go on or something like that. In any case it was just an idea a way of telling you know what the problems of finding and locking are.

(Refer Slide Time: 46:05)



Let us take a more real a real example of a web server alright. So, what is the web server? Web server is let us say you know this running on this machine, which has a disk and it has a network ok. And, a client sends an HTTP request and receives a reply, HTTP response. And, tip I mean let us here let us take a simple case where the HTTP request is a URL and the reply are the contents of that URL, which is an HTML page let us say right.

(Refer Slide Time: 47:13)



So, how is the web server like this implemented? Well let us say, you know at a very high level the web server is probably running a loop like this while 1, so, while true know it is an infinite loop read message from incoming network queue.

Let us say URL is equal to parse message all right, read URLs file. So, whatever is the URL you know you can parse it to forget a file. So, let us say read the URL file from disk right and then right. So, you get the URL file from disk you get the contents of the file and then you write those contents also right as a reply so, you write the reply to outgoing network queue right. So, what am I assuming here I am basically assuming that there is a network queue, right?

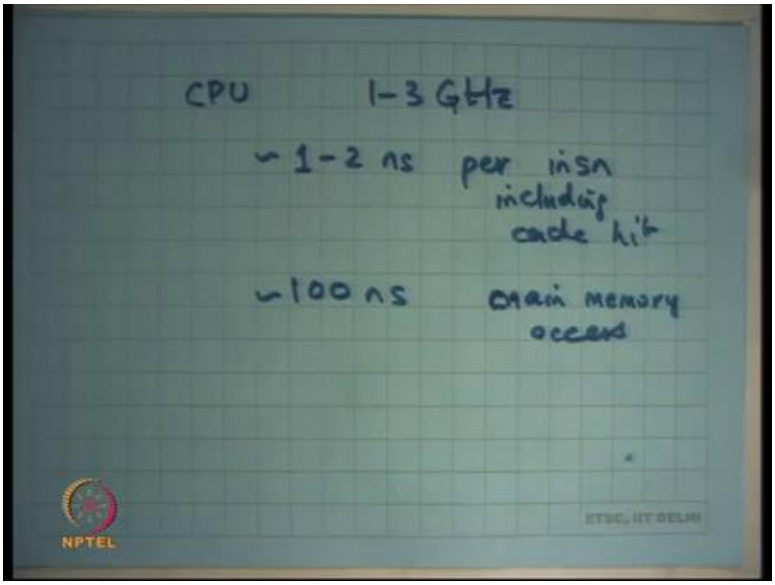
There is somebody who is filling up this network queue. So, there are packets being received on the wire and, those packets are getting stuffed into this network queue incoming network queue. There is this server that is running that is picking up packets from this incoming network queue, processing them in this way and then there is an outgoing network queue, which you just and the server is putting things with an outgoing network queue and there is somebody was paying things up from the neck out doing network queue and putting them on wire right.

So, let us see, what is the performance of this web server all right. So, basically what will happen is let us say the multiple clients in this let us say they are you know there are multiple clients, that accessing this web server their request will get queued in the

incoming queue, and the you know depending on how many clients there are what is the concurrency level of clients the queue will keep filling up and this server will pick up one request and start serving it. So, you know the maximum number of clients that it can serve in a second is depends on how much time it takes to execute this code right.

And, how much time does it take to execute this code? By far the most expensive operation in this is this right. Reading the URL from disk is by far the most expensive operation. These operations are likely to finish in you know 100s of nanoseconds to may be microseconds or something, but this operation URL from disk is an operation that takes milliseconds to complete right. Why does disk take so much time, while the other things are so much faster? Have you discussed this before? No ok.

(Refer Slide Time: 50:13)

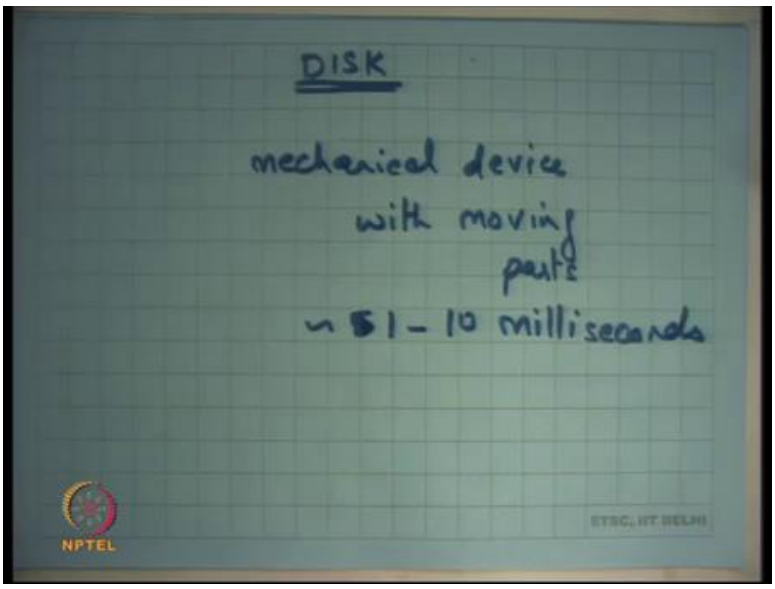


So, let us say there is a so, typically today's modern CPU runs at 1 2 you know, let us say 3 giga Hertz right or roughly 1 to 2 nanoseconds per instruction so, 1 to 2 nanoseconds per instruction. If the instruction was a memory access and the memory access is a cache hit, then also you know typical execution times are 1 2 3 nanoseconds.

So, cache hit including cache hit, if it is a cache miss then you know let me put approximately and roughly 100 nanoseconds for a cache miss or let us say main memory access right.

These are all electronic operations these are just you know semiconductors exchanging electrons to basically access either cache or memory or things like that, the only reason memory is sort of more costly is because you have to travel a longer distance. You have to go over the bus there is some bus contention that you to worry about and then you get to the memory and then you know, but it is all electrons traveling and so, it is very fast right. On the other hand, you know a disk access or a magnetic disk which has persistence is a mechanical device right.

(Refer Slide Time: 51:35)



So, a disk actually if you look at a disk, then it is a mechanical device so, it is a mechanical device with moving parts ok. Exactly what is the structure of a disk and why and so, you know hence it is much more costly and it is on the order of you know 5 or let us say you know 1 to 10 milliseconds ok.

So, that is 10 to the power a million times slower than an instruction access, it is a or which means that accessing a disk operation in that time you could actually have executed a million instructions in CPU. So, we going to discuss how our disk is organized exactly and what determines whether what the access time is exactly. And, then what does it mean for a web server and it is scalability which means how many concurrent clients can it support and how you can optimize it and what role does multi-threading have to play in optimizing it all right. So, we are going to look at that ok.

Thanks.