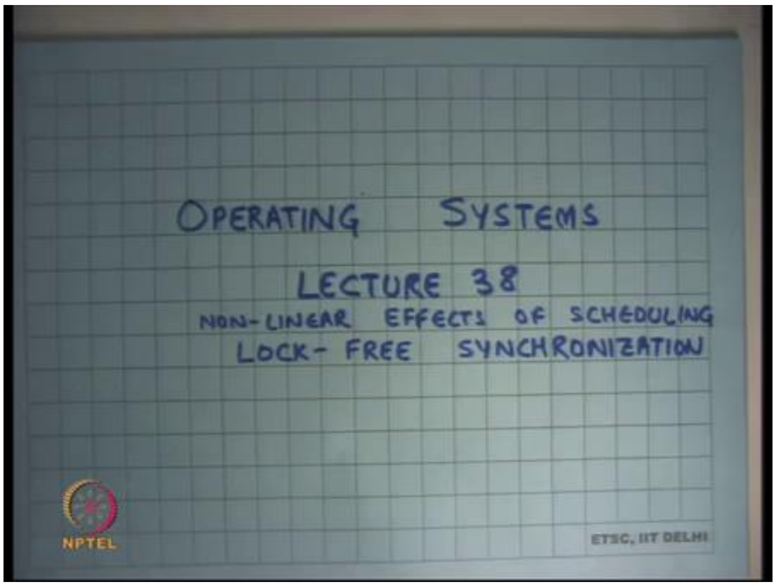
Operating Systems

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

Lecture - 38
Lock-free multiprocessor coordination,
Read-Copy-Update

(Refer Slide Time: 00:24)



OPERATING SYSTEMS

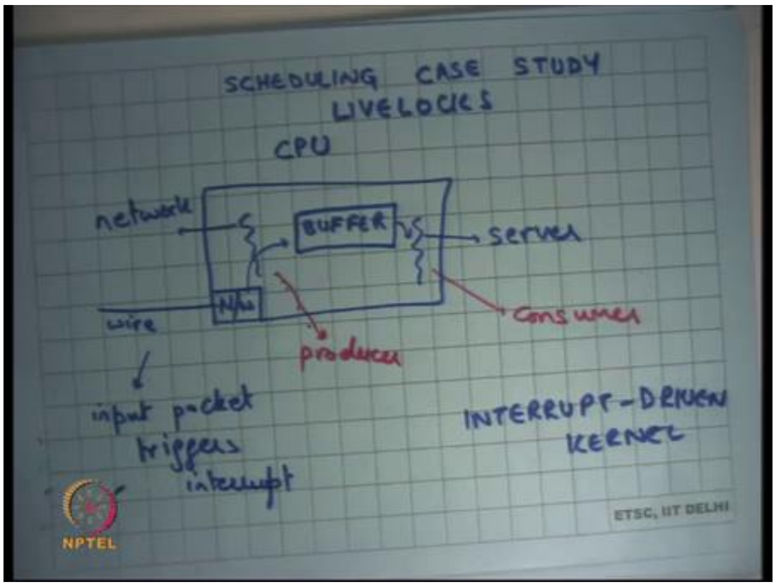
LECTURE 38

NON-LINEAR EFFECTS OF SCHEDULING

LOCK-FREE SYNCHRONIZATION

Welcome to Operating Systems Lecture 38. So, I am going to first discuss continue our discussion on Scheduling, I am going to discuss an interesting case study where we will see that badge scheduling can have very non-linear effects in terms of performance. And, then we are going to talk more about Lock free synchronization after that right.

(Refer Slide Time: 00:44)



So, we have been looking at scheduling and what are the different scheduling policies and we talked about priority scheduling and we said that typical general-purpose systems use some sort of priority scheduling. Where they prioritize IO bound jobs or interactive jobs over compute bound jobs or long running batch jobs and that ensures that there is a high throughput and also very high very low response times right. But these priorities are not strict in nature in the sense that priorities keep changing.

So, if a job has not been received attention of the CPU for a long time, then gradually it is priority will increase and so eventually it is priority will become maximums and there is no starvation problem. So, in general purpose systems you avoid starvation problems of this kind, but in other systems like real time systems let us say you are running an operating system in a car, there you will want to implement strict priorities.

So, some processes are strictly higher priority than other processes. For example, safety related processes will have higher priority than you know other processes that are not so critical ok. But I am going to talk about one case study which will show which will you know give you a nice insight into the non-linear effects of scheduling.

So, if you do a bad scheduling then actually your system can behave rare very very poorly under high load. So, we also said that scheduling is a problem when the resources are saturated in their you know; so, if the number of resources are short for the amount of or the rate of requests that you are getting that that point scheduling becomes really important. If the number of resources is plentiful then scheduling is less of a problem of course alright.

So, let us take this example let us say this box is a computer or you know CPU and let us say this it is connected to the network and this is a network card let us say network device and you know packets are coming in. So, it is a let us say it is a server and their packets coming in from the wire and this let us say it is a web server or you know some kind of a network server and it is receiving packets and then it is sending a reply. So, it is receiving a lot of requests and then sending replies to those requests.

Now, you know one way or one of the typical ways you would implement such a server is that you will use interrupt ways mechanisms to handle incoming packets. So, as soon as you receive a packet you know you will configure your network device or a network card to say interrupt me whenever you receive a packet right. Why do you say that? You know so that as we know that two ways to handle IO, one is interrupt based IO and the other is polling based IO.

So, there are two options here one is I could say I could configure the network card to say every time you receive a packet interrupt me right and interrupt is looks like interrupt is nothing but a you know very high priority processor. So, in interrupt basically preempts all other lower priority processes, assuming there was no other you are not already inside an interrupt handler; whatever was running will get preempted and you will get to run the interrupt handler. And the rationale behind that is that it if a packet that has come in you want to serve it immediately right.

So, there is a request that has come in you want to minimize the response time of the request. So, the reason you would want to have an interrupt-based system is to minimize the response time. The other approach you could have taken was not configured the network card for interrupts and rather said I am going to check the network card every few milliseconds or you know some granularity. So, let us say I checked the network card every 100 milliseconds.

So, in which case you know the network card needs to have a buffer and packets are coming in then the buffer you know it does not need to get buffered in that buffer on the card. So, it has nothing to do with the CPU there has to be a buffer on the device itself and then the CPU can check the network card for packets. What is the problem here is that the response times are higher than what they can be?

If a packet comes on average it will have to wait on the buffer for 50 milliseconds, you know if you are if I am pulling every 100 milliseconds then my response time average response time will at least be 50 milliseconds which is very large. You know I could have done much better if I was using interrupts, then I could have given much, much better response times on the order of microseconds let us say alright.

But so let us say I have an interrupt driven kernel. So, I am going to say it is an interrupted driven kernel right. So, which means that each time I get a packet I get an interrupt and the packet gets copied from the network card to the Buffer, to some buffer that I am maintaining in the main memory right. And, then there is another process server which is which is consuming from this buffer and doing some processing on it. Let us say you know it is formulating a reply and sending a reply back on the network card or some other network card doing something with the packet right.

So, with an interrupt driven kernel you know I modelled this as a producer consumer, where the producer is the network card and the consumer are a server thread. And, an interrupt driven kernel the producer is basically always a higher priority process than the consumer. So, if the producer has something to do for which means it has some packet to produce inside the buffer, it will get to run, and it will preempt the consumer every time right ok.

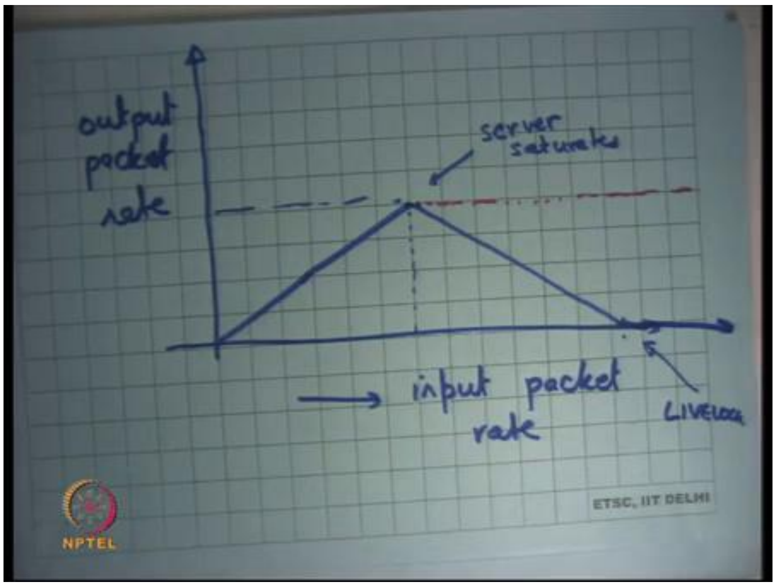
So, this works this has very low response times because, you know you will basically immediately produce and consume and things like that. But, let us see what happens if there is a very high load on the network card. So, let us say now I have a very high-speed network link and my CPU cannot process links the packets at that rate right. So for example, you know I am receiving packets at the rate of you know on I am using a 10 gigabyte per 10 Gbps link and I am getting lots of packets and the processing I need to do on the CPU is cannot sustain that rate of packets ok.

So, what is going to happen in this case? The packets will start getting dropped inside this buffer. So, this buffer will fill up and the packets will get start getting dropped at this buffer alright.

What will also happen is that because you are interrupting on each packet and the interrupt rate is so high, because you are getting packets at a very high rate the server thread will never get to run. So, the consumer will never get to run right. So, that is only the producer that is getting to run, and the consumer is never getting to run, which effectively means that the CPU is actually doing no useful work.

All it is doing is getting the packet from the network card to your buffer and there is no follow up on the buffer and eventually for the packet to get dropped. So, you know this act of actually bringing the packet from the network card to the buffer is completely wasteful in this case right. Because, that packet is destined to get dropped eventually right or in fact you know if the buffer is full then it will just get dropped right there right.

(Refer Slide Time: 07:57)



So, this is an example of a very bad scheduling right. So, what will happen is if I was to draw a curve between let us say if I wants to draw a curve between input packet rate and let us say output packet rate. Let us say my server was sending a reply for every packet right. So, let us say there was an input packet rate and output packet rate, this curve will be were pretty much linear for low rates. But, at some point when the input packet rate is so high that the server actually gets saturated, you will basically start seeing a drop in the throughput till it you actually reach 0 right.

So, this is the point where the server saturates and the reason you are basically seeing a drop is basically because, after the input packet rate becomes higher than a certain limit then the producer gets to run more often and the consumer gets to run less often. And, so because the consumer is getting to run less often you know the net throughput is dependent on the consumer right which is the bottleneck. And, so the higher the input packet rate the less often the consumer gets to run till the consumer does not get to run at all no.

So, the input packet rate becomes so high that the consumer does not get to run at all and at admit point you basically have a net output rate of 0 right. So, this is the point at this point we say it is a live lock system right, we call a system live lock if the system is doing a lot of work which making look progress right. So, here is this is definitely a case of a situation where the system is doing a lot of work it is continuously the CPU utilization is 100 percent, yet the output net useful output of the system is actually 0. So, what is the problem how could I fix this problem?

Student: If the rate is very high, we can switch to polling.

If the rate is very high, then I could have switched to polling and so you know what is the best good answer and so what is the best I can expect. So, let us go in the reverse direction. So, I do not like this fact that is the input packet rate is. So, high then my throughput becomes 0. What is what would have been the ideal system?

Student: Freezing and then saturating.

Right, so the ideal system would have been that at this point the CPU got saturated, clearly I cannot serve anything which is more than that after all the server cannot do anything more than that, after this I should have had a curve like this right. So, after that you know all the packets that are coming no extra rate is getting discarded. But the rate that I can support at least that much is getting served right, that is the ideal thing you could have had in your system right and why am I not being able to achieve this ideal behavior.

Because, I am doing bad scheduling right because I am doing bad scheduling because I am prioritizing the producer over the consumer irrespective of anything else right and I did that because I wanted very low response times right. So, here is a very nice example where because of trying to minimize the response times I actually get a very bad throughput in my system alright. So, and so what is happening is that only the producer of the queue gets to run, and the consumer never gets to run and so you basically degrading your throughput. So, how would you reach this point from this point, you would want to reprioritize things you would have want to say that ok.

You know I can see that there is some problem you know I want to give equal priorities to my consumer and producer at some point. It should not so happened that producers getting to run too much, and the consumer is not getting to run at all right or but consumers getting to run less than the producer, that to me is a warning sign right. That is if the consumer is getting to run less than the producer or in other words if the consumption rate is less than the production rate then that says it looks like a warning sign and what I should probably do is throttle my producer. So, that you know the net throughput remains nice, otherwise if the producer just running is just doing wasteful work right.

So, the idea is this thing that could have happened is that instead of the packets getting dropped at this buffer, the packet should have been dropped at the network card itself right. In that case the CPU will not have done any wasted wasteful work right what is happening is that the work that is required to bring the packet from the network device to the buffer is completely wasteful in the case of we know when you reach the live log stage.

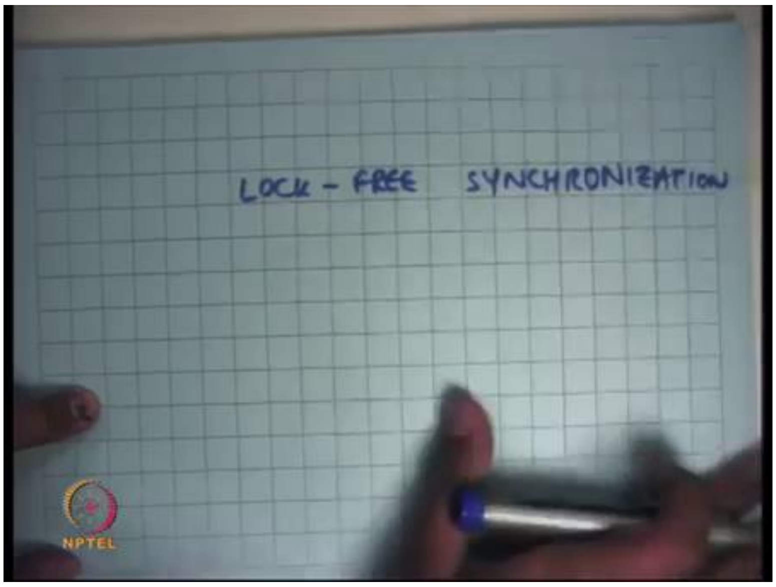
But, if you somehow figured out that no you know I do not need to do this wasteful work, then you could then what would happen is eventually the packets would get dropped here right. And, one way to think about this is that you know if you notice that your buffer is becoming higher you know if the buffer lengths average buffer lengths are becoming higher than a certain threshold. Then you throttle your producer throttling your producer basically means that you switch from interrupt mode to polling mode and you have to choose your polling frequency right.

The nice thing about switching to polling mode is that you choose you can choose your polling frequency and the choice of polling frequency determines the rate of execution of the producer or the priority of the producer right or the proportion of the producer. Eventually I want that the producer and consumer rates should be similar and so you know that is how I would ensure that kind of thing right.

So, the solution to this says that you know which is which is let us say implemented in mainstream kernels and so this was a problem in the Linux kernel. You know let us say 15 years back it was discovered and so the solution that was implemented after that was that you use interrupts. If the rates are low and then you at some point you switch to polling when the rates are high, and you choose your polling frequencies depending on your average buffer lengths and the time it takes to basically.

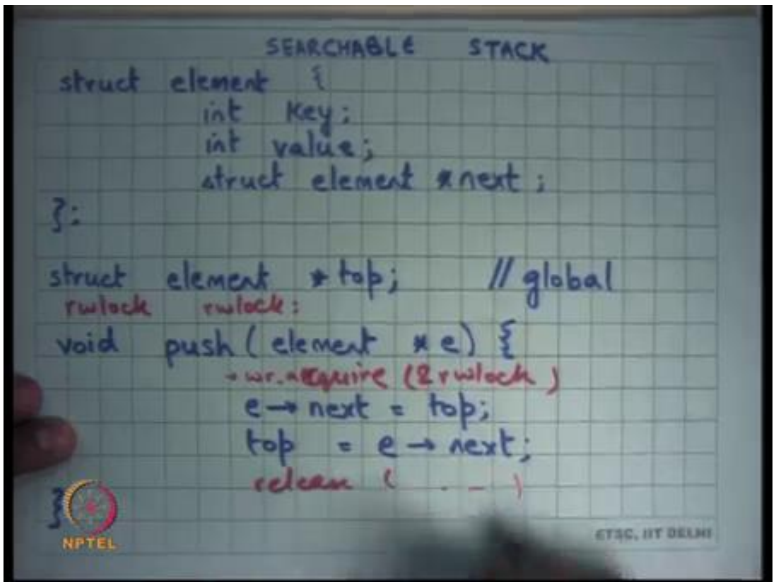
So, the time the producer and consumer processes are getting to run right. So, here is an example where there is a tension between response time and throughput, and you want to have both of them dealt well. So, in general you know the thumb rule is that interrupts are better at low at low rates and polling is better at higher rates right ok.

(Refer Slide Time: 14:22)


LOCK - FREE SYNCHRONIZATION

With this I will switch to my next topic which is Lock Free Synchronization. So, we have discussed some lock free synchronization before, and I am going to discuss more of that today alright.

(Refer Slide Time: 14:35)



SEARCHABLE STACK

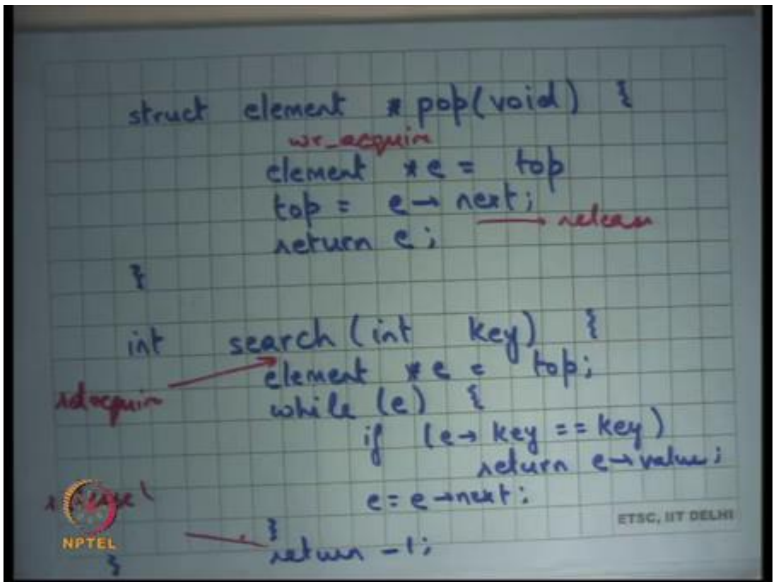
```
struct element {
        int key;
        int value;
        struct element *next ;
};

struct element *top;      // global
 rwlock    rwlock:
void   push ( element  *e) {
           wr.acquire (&rwlock )
            e→next = top;
            top  = e→next;
           release (  . _ )
}
```

ETSC, IIT DELHI

So, let us and let me first introduce to you a small example, let us say I wanted to implement a data structure called a Searchable Stack. The searchable stack has elements of this type an element is a key and a value, and a next pointer and the stack is represented by a variable called top right.

So, you can only look from you can only start at the top, you push to the element by just incrementing talk basically you wanted to push an element e you just say e.next is equal to top, top is equal to e.next (Refer Time: 15:10) that is pushing an element onto the stack.

(Refer Slide Time: 15:13)



```
struct element * pop(void) {
        wr_acquire
        element *e = top
        top = e→ next;  ——→ release
        return e;
}

int     search (int key) {
rdacquire ——→ element *e = top;
        while (e) {
              if (e→ key == key)
                     return e→value;
              e= e→next;
        }
 ——→    return -1;
}
```

And. you top an element by just decrementing top you say e is equal to top, top is equal to e.next and e right. Of course, I am gliding over some details like f top is not equal to null and etcetera you can do that. But there is a third operation which is a search which just goes through the stack and searches for a particular key.

So, you want to search for a key you just iterate over the stack while e fe.key = then return you got value otherwise just do e.next, if you find it very good if you do not find it will turn this right. So, there is a searchable stack the three the two operations push and pop which are write operations read write operations. And, there is one operation which is a read only operation called search you know this search is the search as stack clearly in a if this code is executed concurrently, if multiple threads are sharing the stack searchable stack then there is a problem right.

I am going to discuss that we all know there are concurrency problems, in this course there is a concurrency problem if somebody is searching and somebody is pushing at the same time and also there is concurrency problem. If somebody is pushing the two threads are pushing at the same time or pushing and popping etcetera. So, what are some ways to solve it well one way to solve it is usually use locks right. So, you basically say I have a lock for the entire data structure. So, and basically put a lock acquire and release around this you put a lock acquire and release around this and you put a lock and acquire and release around this, that is a coarse-grained lock right. And that will severely impact your performance it will not scale at all.

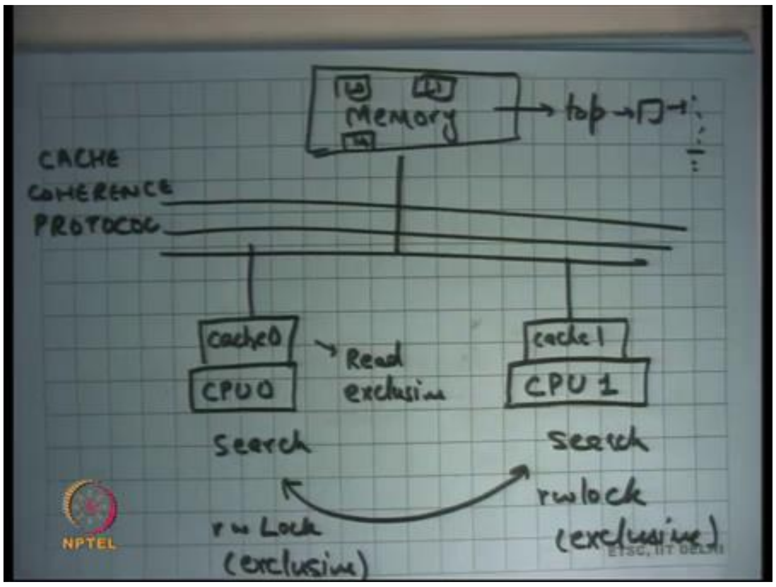
So, if you have multiple threads then you will only have single threaded performance, let us say this code was the only code that is running then you will you basically have no scalability. What is the other thing you can do let us say I tell you that most of the times you are going to execute search and push and pop are relatively rare operation?

So, you know just as an example 99 percent of times it is search that gets called and only you know one percent of times it is either push or pop that it is called. So, it is mainly search that is getting called and search is read only operation. So, what will you do reader writer lock right? So, simple I will just use the reader writer lock, I will say you know I will have I will have.

So, I will declare a reader writer lock let us say rwlock and then I will have write acquire rwlock and release and I will have similarly I will have write acquire here and I will say release and here I will say read acquire right and I will say release here. So, why is reader writer lock better than your normal lock because the semantics are that multiple the lock can be acquired in read mode multiple times simultaneously.

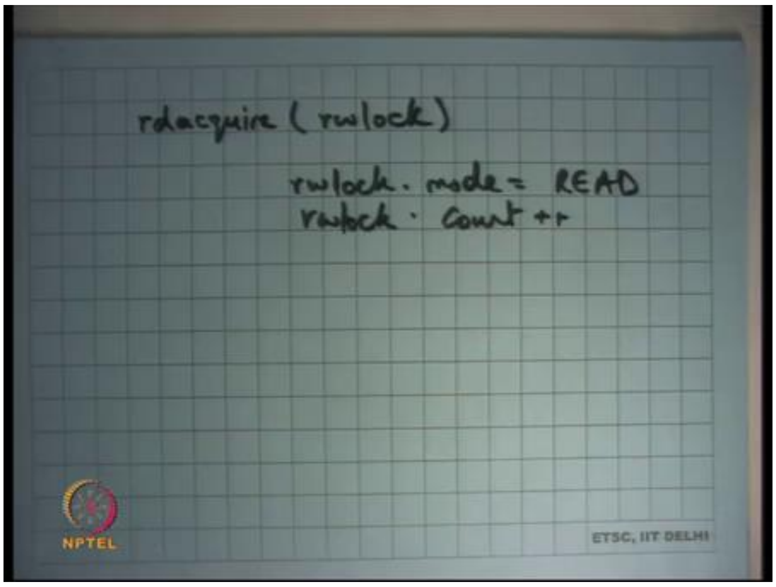
But the lock cannot be acquired in simultaneously in write and read mode or simultaneously in write and write mode right. So, there can be ninety nine percent of times you are calling search and all these threads can execute simultaneously. But is it really better than is it really that good? So, let us look at what is really happening at the CPU level or at the hardware level ok.

(Refer Slide Time: 18:43)



So, let us say you know I have CPU 0 and CPU 1 and let us say I have a shared bus and I have caches here. So, have cache is 0 and I have cache 1 right and we have connected to the bus in this memory ok. Before I discuss this let us let us also revise how reader writer locks are implemented.

(Refer Slide Time: 19:20)



rdacquire (rwlock)

rwlock . mode = READ

rwlock · Count ++

NPTEL
ETSC, IIT DELHI

So, you know if I want to implement a reader writer lock, I will say you know how will I implement read acquire you know I will say I will set some flag and say rwlock.mode is equal to read and count++ right. Something of this sort and you know I must make sure that it is not already held in write mode right. So, I am going to and so this I am going to have to write to this lock variables I am going to I have to maintain a state in this lock variable that it is read it is held in read, read mode. And I will have to also write a count variable it says you know I have I have basically acquired it in read mode.

So, that other readers know that, or other readers and writers know that acquired. So, what has happened is ah? So, what I am going to show you is that this code which was actually completely read only code has now become a read write code right. So, here I am only doing read operations on all the shared values. What does my shared value, the shared value is my stack and all I am doing is I am just doing reads on the stack right. The only rights I am doing is to my local variables which can be you know my local stack or my local registers, whatever but that is not that is not shared so I do not care about that.

So, the only things that I am the only operations that I am making on the shared variables are reads, but the moment I put a read acquire and a write acquire these operations involve a right to a shared variable all right. And that is a bad thing and so why is it that a bad thing? Because if I look at the hardware level, if there is shared data let us say there is there is a stack living here let us say this you know top this is the searchable stack living here and so on. Then if I was if both of these were calling search, then very likely this stack would get cached in the local caches of both these CPU.

And so, these CPU can just execute search at cache speeds, they do not need to go to the they have bus to be able to read any memory locations right. Assuming that all local variables are already in the cache or in the registers, the global variables if the access of the global variables are read only accesses then you know modern hardware allows you to cache both of them simultaneously in read mode only right. So, before so just some background: hardware implements what is called a cache coherence protocol, how many of you have studied a cache coherence protocol not really alright.

So, a cache coherence protocol basically says, you know it allows so for every memory location m memory location M can be cached here. So, it basically maintains coherence of data which means that a memory location M can be cached here in read mode or exclusive mode right. If another cache accesses it also then if the cache memory location is held in read mode, then this can also cache it. And so, the same location can be held can be cached read mode and multiple caches simultaneously. But a location can only can only be cached in exclusive mode in one cache at a time right this is clear.

So, every location can either be cached in read mode or in exclusive mode, if you know a a location can be cached in read mode in multiple caches simultaneously known to increase performance and because it is in read mode nobody can modify it. If somebody starts to modify it what happens is all the other read mode cache entries get invalidated and this the one that modified it now holds it in the exclusive mode alright. So, it says that and this mechanism to implement this kind of protocol is implemented in hardware and it is called the cache coherence protocol right.

So, if the CPU are accessing the list in read only mode, what will happen is that this list will get cached in the local caches of the CPU in read mode eventually. And what will happen is that all the search operations will only access the local caches and never have to go to the bus right. Recall that you know cache speeds are on the order of few nanoseconds in one or two nanoseconds. On the other hand, a bus transaction involves depending on you know whether you actually go to the bus or whether this scores are within a single chip, it can range from tens of nanoseconds to hundreds of nanoseconds.

So, it is on that order, so it can be 10 to 100x slowdown from just cache accesses right. So, if you just if you only have cache it is you versus if you do not if you only have cache misses you can have a 10 to 100x performance difference in the thing. So, earlier you all the things were getting cached in the local caches and so both the CPU could have executed search at full cache speeds.

But now because you have a read acquire and write acquire there is another shared variable here which is the lock right. And because the lock is being accessed in read write mode it is been modified, what will happen is that the lock will get cached here will need to get cached here in exclusive mode. Then this if you access try to modify it and so it will get need to get invalidated here and now it will get cache here in exclusive mode and.

So, the lock will keep bouncing the read write lock will keep bouncing between the two caches right and because it needs to be cached in exclusive mode, because it is a write access to these variables right. So, each time you access if you each time you write to the read write lock you actually need to invalidate the value in the other cache if it exists to get it to get the latest value right.

And so, what will happen is that this read write lock will keep bouncing between the two CPU and each time it bounces it involves the bus transaction and that gives you a lot of slowdowns alright. So, what can happen is that this code will actually becomes it is possible that this code actually becomes slower with read write locks then faster right.

So, for example, if you have two CPU and if I tell ask you to implement the fastest possible you know fastest possible a solution to my program which involves accessing the read write lock. You may say let me use a read write lock, but if you use a read write lock because of the cache line bouncing the total performance of two CPU may actually be lower than the performance of one CPU.

Because if you only ran the program with one CPU; it would have been a cache line access cache access. But, if you access a two CPU if you paralyze the program with read write locks even though there is full concurrency, because of cache line bouncing you have actually slow downed the program by a factor right.

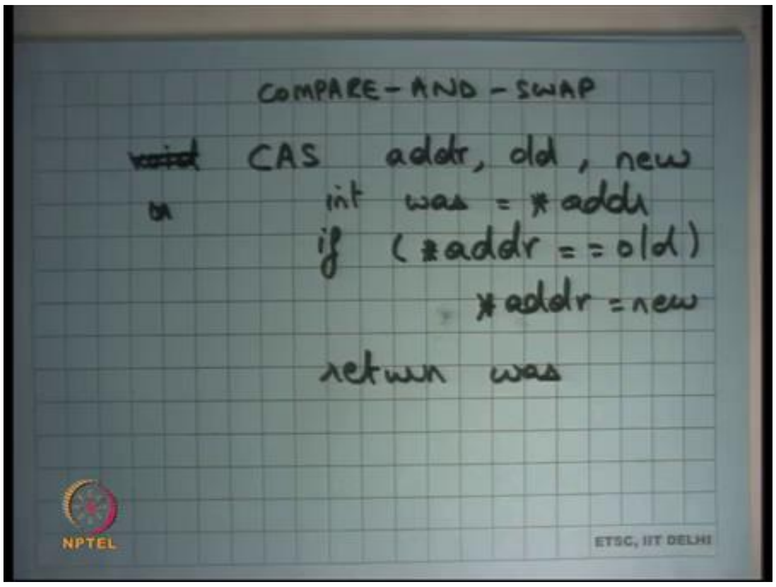
So, the advantage of actually having more concurrency is gone in some sense right. In other words, you even though you know the CPU are called are executed concurrently the bus becomes a serialization bottleneck in some sense all right. So, what was the problem, the problem was that I had a read only function and by using read write locks I converted it into a read write function and that in that was that made it less scalable right.

This problem I have demonstrated this problem on two CPU, but this problem becomes even worse actually much worse if you go to larger number of CPU. For example, you know today you can get a machine which has eighty CPU in it you know a proper and you know a full desktop machine desktop processor like the Intel for example.

And so you can you can get a machine with an eighty CPU machine eighty CPU eighty CPU on it and then you have a cache coherence protocol going on between the eighty CPU. And, now this lock is bouncing between the eighty caches and so that can become a huge performance bottleneck all right. So, firstly it clearly shows that you know reasoning about performance and concurrent programs is extremely hard.

But what could have what could I have done, basically it is a bad idea to convert a read only code to a read write code right. It would be nice if I could not, I would not have I could have done this synchronization without having to do this right. Why do I need to have this read write operation inside my read only function, because I need to synchronize it with other write operations right that is the only problem I have? So, what could I have done right let us say I used a lock free primitive like compare and swap?

(Refer Slide Time: 28:03)



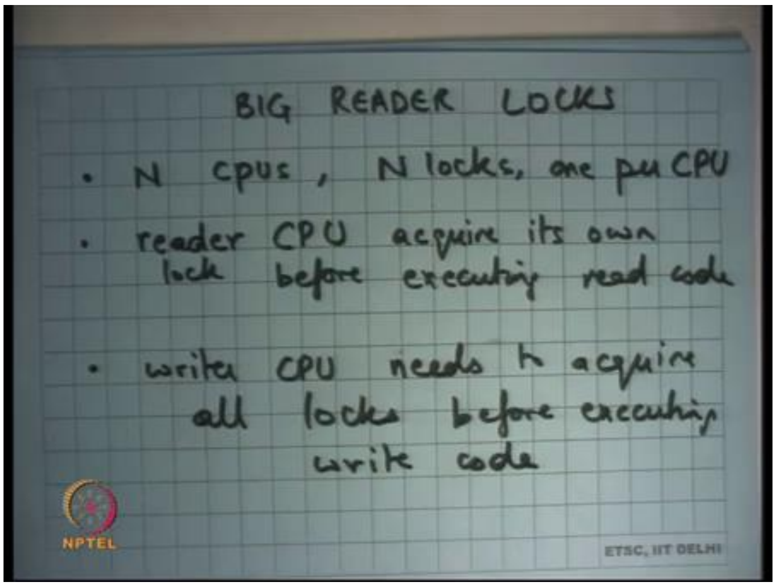
COMPARE-AND-SWAP

```
~~void~~ CAS   addr, old, new
[illegible]     int was = *addr
                if (*addr == old)
                        *addr = new

                return was
```

NPTEL
ETSC, IIT DELHI

So, we have seen compare and swap before and I said that I am going to use compare and swap to implement my writes and because my compare and swap is going to be atomic my reads do not need to take a lock let us see if that if this works alright. So, what I am going to do is I am going to implement ok. Before I talk about compare and swap let me talk about one alternate solution.

(Refer Slide Time: 28:33)



BIG READER LOCKS

- N cpus, N locks, one per CPU
- reader CPU acquire its own lock before executing read code
- writer CPU needs to acquire all locks before executing write code

So, here is one alternate solution what are called br locks Big Reader Locks alright. So, big reader locks have this let us say there are N CPU, or you know you can generalize it thread, but I am I am going to consider N CPU. Then you have you know you have N locks one per CPU, reader CPU I know I am just using a term thread and CPU interchangeably. Reader CPU acquires it is own lock and before executing read code and writer CPU needs to acquire all locks before executing write code ok.

So, idea is that the, so you have a lock which is a large you know which is a structure which has now instead of one bit or two bits it has N bits. And the idea is that every CPU will acquire it is its own lock it is own bit and the other CPU, so and the writer CPU will need to require all the bits right. So, why is this better than the reader writer lock that we just discussed let us see. What will happen is that the let us say the CPU 0 and CPU 1 so you have two locks CPU 0 will have it is own locks.

So, there are two locks L0 and L1 CPU 0 will make a write operation, but it will make a write operation always to the same location L0 right. And so, what will happen is L0 will get cached in cache 0 and L1 will get cached in L1 both an exclusive mode. But there will never be any bouncing between the caches. So, L0 will never need to go to L1 and L1 will never go to get need to go to L0 and so they will both live on caches and yet you will basically ensure that it is correct right because you know.

So, readers can execute concurrently because you know one CPU has acquired L0 and other CPU acquired L1. But a reader and the writer cannot execute concurrently, because the writer will acquire all locks. So, you know it will writer will need to acquire all locks. So, what you have done is you have slowed down the writer, but you have really made the reader faster right. And that is the case we are optimizing for I am saying that the reader is executing ninety nine percent of times and writer I do not care about the writer performance at all ok. So, this works what are some problems.

Student: Sir the writer can start.

The writer can start well, not really. I mean it just depends on you know. So, it just depends on your lock acquisition algorithm. So, you can just say you know I am going to give locks on a first come first serve order. So, just like a reader writer lock you can say that you know, if some if a writer makes a request for a lock li then you know it will eventually get the lock within a finite amount of time within a bounded amount of time let us say.

It is the writer cannot stop, but the writer actually has to make a global operation if you are sixty you know if you have eighty CPU and tasks to do a lot of a lot of work to get there. The more important thing is that these caching is not done at bit granularity right caching is done at cache line granularity right. So, you basically just so just like pages, pages are a unit between disk and my main memory, the unit between a main memory and the cache is a cache line right.

You cannot just say I want to cache this byte you have to say I have to want to cache this line cache line that contains this byte right. So, it is a full cache line that you have to get into the cache right. So, what does that mean for our big reader lock can li means sharing the same cache line no right. So, all these different reader locks n locks need to be on different cache lines in. So, basically you will need to make sure that the variable li is occupying one full cache line right.

So, typical cache lines let us say will be 64 bytes on modern hardware you need 64 bytes for every CPU. So, if you have 80 CPU and you have 64 bytes into 80, so the lock structure actually becomes pretty large and that is it right. Moreover, you know you have to you cannot write generic code it literally depends on what the cache line is on the architecture. So, you basically need to check what the cache line is and compile your code for that particular for that particular chip right.

Different chips for the same architecture can have different cache line sizes and you need to optimize based on that. So, these are all you know difficult things to do. I mean imagine if I was to use big reader locks and the Linux kernel then to sort of worry about which chip is this kernel going to run on to basically decide what say structure of my big reader lock should be so.

So, this is not an option, but, but let us look at a better option you know which does not have these problems and I am going to try to use compare and swap to basically implement lock free synchronization and in this case example in a better way right. So, what I am going to do is. Firstly, why do I want to void locks, let us just also review why we want to void locks.

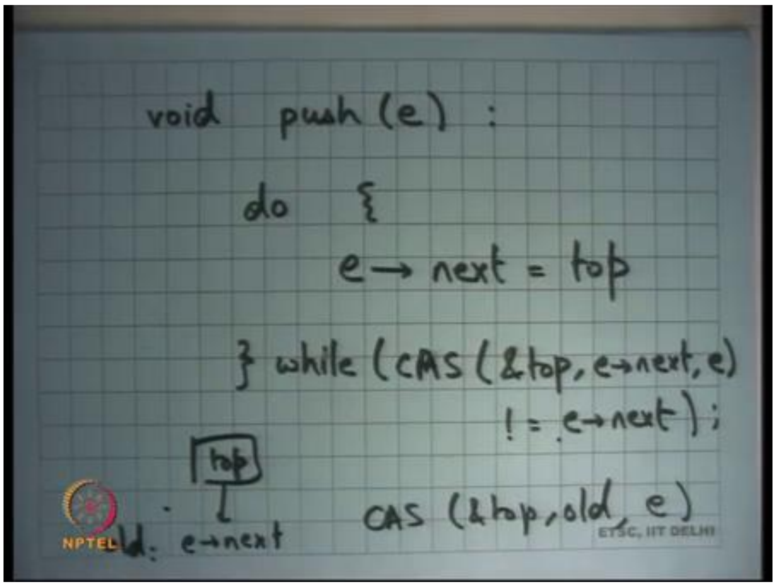
Firstly, I have already discussed locks performance problems you know the cache line bouncing, even for read only code even if I read use reader writer locks. There is complexity involved in locking you have to worry about you know fine grained locks for this coarse-grained locks etcetera, you have to worry about deadlocks, and you have to worry about priority inversion ok.

So, we have also studied scheduling and there is a problem with locks, locks are just resources and people need to debate on resources. So, you know anytime you have something like that then this priority inversion that is a problem ok. So, you know let us look at the let us look at the other way of doing locking synchronization is compare and swap, you know which is which is sort of a transactional way of doing things. We will basically let me implement to push and pop using compare and swap, so let us just revise for what is compare and swap.

Compare and swap is of an instruction let us say let us say CAS that takes three arguments one is an address; another is the old value and the third is the new value. And the semantics are if the contents of address are still equal to the old value then replace them with the new value right. And, you do this in an atomic way also this instruction returns which means it overwrites one of this these registers with the value that was read.

So, let us say I added a value which was you know let us say I said was is equal to star adder and then I return was. So, I read this location and I compare, and I returned that firstly always. But I also compare it with the old value and if it is equal then I replace that location with the new value and this entire equation is atomic right that is a compare and swap you compare, and you swap. If the comparisons are successful, if the comparison is not successful then you do not fall right.

(Refer Slide Time: 36:43)



```
void  push(e) :
      do  {
            e→ next = top
      } while (CAS(&top, e→next, e)
                        != e→next);
      top
old: e→next       CAS(&top, old, e)
```

So, that is the compare and swap and let us see how we use it for something like push. So, I say void push element e I am going to say e.next is equal to top right and then I wanted to do top is equal to e.next. But I wanted to do that top is equal to e.next only if top has not changed in the two between the two are instructions right. So, what I am going to do is I am going to say do while compare and swap top address old value which is what you read e.next and new value e is not equal to e.next right.

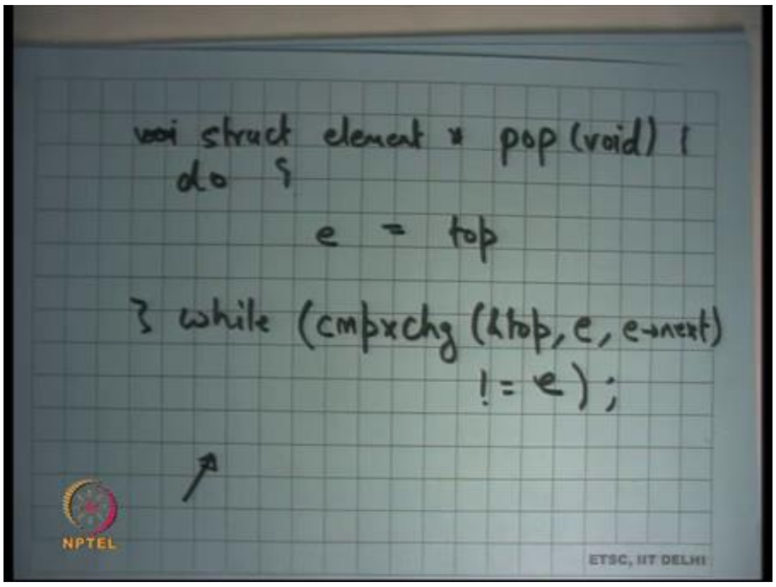
You check atomically if the value of top is still equal to what you read it earlier and if so, you replace it the e right that is what you do right. And, this check with e.next is basically saying whether this was actually successful or not if this was successful then you are done you break out.

If this was not successful which means it was not equal to e.next it was not successful, then you retry this operation. So, let us say this is top I read the old value in e.next this becomes old, then I execute CAS on top and old and the new value that I want is e right.

I want to push the elements I want top to become e so and so I do execute this CAS and top old e right. This will become successful if nobody else has modified top in the middle.

So, you know your top will get replaced with e. If somebody has modified top in the middle then this will become unsuccessful right and the way to check whether it was successful or unsuccessful is to look at the return value that is all and you keep trying until you become successful right. So, this is this how we have to push and similarly you can do pop right. So, I am going to write pop you know you can just this piece of code and pop is very similar. So now, let us say let me just write pop because you know it is going to be important in the discussion that follows.

(Refer Slide Time: 39:17)



```
~~voi~~ struct element * pop (void) {
  do {
          e = top
  } while (cmpxchg (&top, e, e->next)
                        != e);
```

So, let us say l struct element star pop I declare a local variable e is equal to top and then I tried to compare an exchange top. The old value that I have read is e and I replay want to replace it with the new value which is e.next right e.next right. That is what I want to do, and I want to do it in an atomic fashion, it is possible that this operation fails how do I know whether this operation has failed. It has failed if the return value is not equal to e right that is it and you put it in a while loop and that is your pop right.

So, that is your push and pop, what are you doing you basically making sure that atomically the list gets updated either with a push or a pop. If I do that what happens to search, do I need to can I say that now search can execute in a completely unsynchronized way, search does not need any synchronization. Well I mean what are the guarantees that I want to get to the user.

I want to get to the guarantees that the user that my data structure always remains well formed right and my search always sees a consistent data structure, which means if it starts from a top then it sees a location, that now there is the serializability in the access right. So, by serializability me it means there were about ten operations that are given to you, then the final results that I get for all those ten operations look like some serial order like it seems like there is some the operations were done in some serial order.

So, either in a push was done first and then there was a search then was a pop then there was a search and so on or you know something. So, there must be some serial order they should be so that is what the guarantee I want to make. So, the whatever final result I get it should obey some serial order there should be some serial order, that basically will give the give you the same result as you as what you saw.

So, in this case because you are automatically updating the pointer the search if it executed before the compare and exchange will see a list where the push had not been made and if it executed after the compare and exchange it will see a list it such that the push has been made right. So, it serial it is the serial order depends on whether the search executed before or whether the search started you know whether you started with the top pointer before the push or after the push.

So, when you read the top pointer, whether that read of the top pointer was before the push or before the compare and exchange before the CAS or after the CAS right, that is what we are going to make it serial and that is going to define the serial order. So, if it was after the CAS then it is as though the search executed after the push, if it was before the CAS it is as though the search executed before the push ok. But that is not all so but let us say you know; I am holding a pointer to so firstly you know let us see what are some problem with this kind of a compare and swap.

Firstly, why was I able to do this, I was able to do this because my data structure was such that updating it required one pointer swap right. I was pushing involved swapping between e and e.next and popping involved for invading e.next and e you know something of that sort and so I could do this.

But if my update operation required swapping two pointers or three pointers or n pointers then I cannot use CAS right then I use need. So, let us say for example, I want if I wanted to support another operation it said remove an element in the middle of the searchable stack right. So, I give you a pointer inside to an element inside the middle of the stack and I say remove this pointer element.

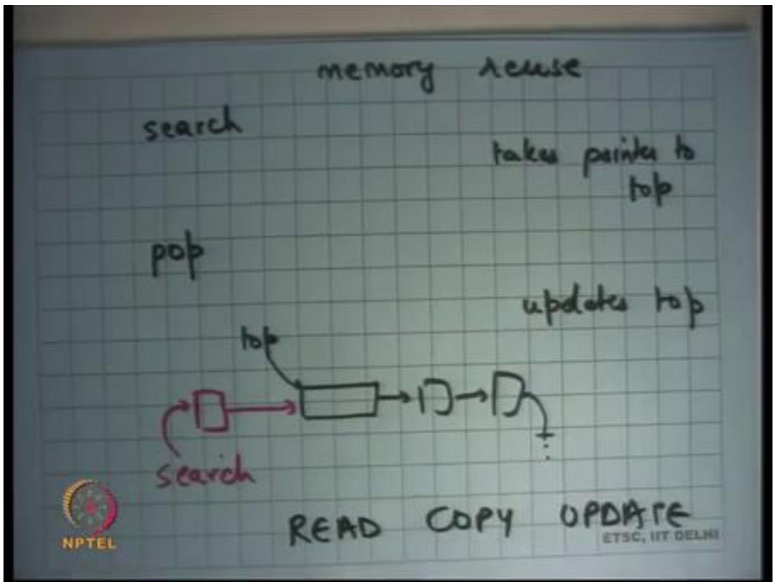
So, removing that element would involve changing the pointer from you know let us say w link list, then I would need to change the pointer from a previous node to the next node and from the node next node to the previous node. So, I need to make two pointer changes right and I cannot do it atomically using the CAS instruction. If my hardware in supported a double CAS instruction which in what they took 5, 6 arguments, then yes or 6 operands.

Then yes. I could have supported something like that right x86 or you know in general architectures do not support more than one memory location for CAS. So firstly, you know CAS can be used for if the operations involve only a single pointer update and one needs to be very carefully reason about it. So, that is one disadvantage the yes.

Student: Sir, in this case say if the pop is not a free the pointer

If the free the pointer right good. So, the second problem is memory reuse alright. So, let us say I popped a pointer, so I popped the locations on the stack right and there was a concurrent search that was running. So, let us say let us draw a timeline.

(Refer Slide Time: 44:41)



So, search gets to execute takes pointer to top right, then pop gets to execute updates top right. So, we said it is because you know it is as though the search executed before the pop and so as long as the top pointers will still exist, and the top is still pointing to the to the disk.

So, what will happen is at this point you will have a list where the top is pointing like this and but you also have search holding a pointer, search holding a pointer to a location that has pointing to this right. So, that is what that is what will happen. Search is holding a pointer to something, but you know, and you have popped it off, but you are still holding a pointer.

And you say it is because you know this location is still pointing here and you know when the search is going to execute like this it is still going to see a consistent list. So, it is ok. But it is if this location is not freed right if this location was freed by the pop then this pointer can become invalid right. So, memory re-user is a problem right.

So, I am the search is holding a pointer to a location that is no longer a part of the data structure, search operation will remain consistent if that location is not freed right. So, we have a strange situation there we have a pointed to us location that is not really a part of any global data structure. But yet I do not want it to get freed because some local references to it may still be hanging around.

So, the short answer to the question whether I need to do something to search is that well I mean search will work just like that except that you need to be careful about memory ok. So, you cannot just free the location. So, question is when can you free the location, you are executed popped this location is no longer used when can you call free is there a guarantee to when I can call free.

Student: If there is no if there is no pointer which references it.

If there is no other thread that holds a reference to this location how do I know if there is any other thread that is holding a reference to this location.

Student: Sir again this struct element we can keep a counter of how many location.

Wonderful, so here is an here is a suggestion that let us put inside the pointer inside the struct element, let us put a reference count right which says how many people have hold I have holding a reference to it right is that a good solution.

Student: (Refer Time: 47:41) which lock kind of.

Yeah, it will have the same cache line for bouncing problem right. So, each time you search called search you have to increment the reference count and so you have made a write operation out of a read operation. And, so reference count is a possible solution, but it is you know it brings us back to the same issue that this cache line bouncing with the reference count right.

So, the answer is really I cannot I cannot have an you know, if I just want to if I do not want to have read write if I do not want to have a write operation global write operation. Then it is very difficult to know whether there could be any thread holding a reference to a location that that is no longer part of the data structure.

So, I do not have any answer and when I can free it well. I could say something like the following though I could say you know let us wait for one hour and then free the location right. That makes sense because after all if there was a search procedure that of holding a reference to it, it must have you know gone you know we have moved on with it you know how long can it hold the location to the search. But, that is not a good answer right because after all it can hold a reference to this location for 1 hour, you know there is no guarantee that it will not it leave that location after 1 hour and that how did I choose one hour why is 1 hour is a good numbers etcetera ok.

So, we are going to see you know we are going to see a solution to this problem in the context of the Linux kernel, which is called read copy update and we are going to discuss that next time ok.

So, let us stop.