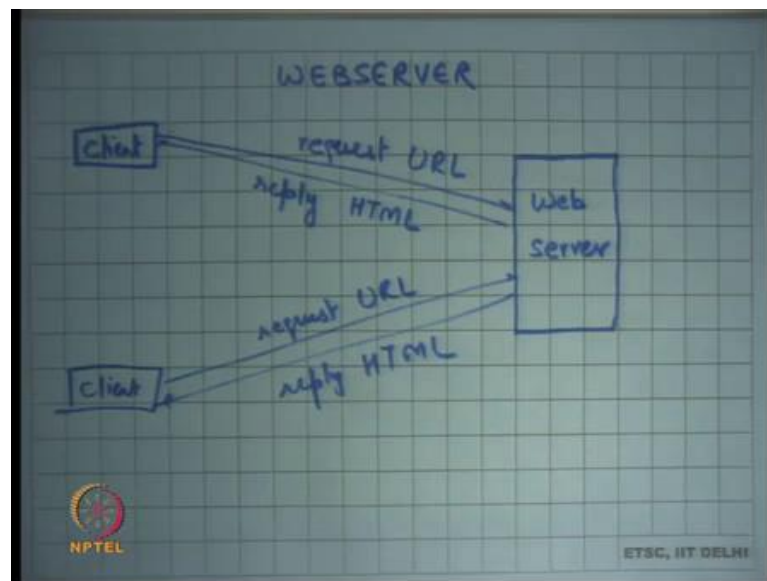


Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 24
Condition variables

Welcome to Operating Systems lecture 24 right.

(Refer Slide Time: 00:30)



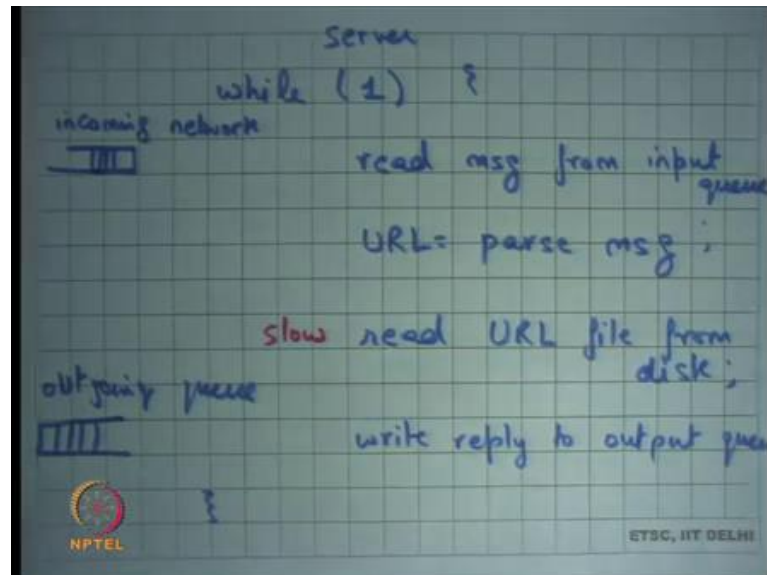
So, last time we were talking about locking, we were talking about fine grained locking, what kind of problems we can run into while doing fine grained locking. And how deciding which locks to use, and where to put the acquire and release statements it is a bit of an art, and one needs to do it.

By looking at the program as a whole and we were looking at this toy program which was maintaining bank accounts etcetera. And I am going to take talk about a more realistic example today which is a web server. And let us say you know web servers implemented on a single machine in our case.

Let us say and there are multiple clients that can connect to this web server. And the clients connect with the network, so client will send a packet which will contain the request which will know for an HTTP server it will be a URL.

And the server will reply with a page which will be the corresponding HTML page right. And multiple clients can connect with this server simultaneously alright.

(Refer Slide Time: 01:28)



So, this is you know the pseudocode of the web server you have an infinite loop. And this infinite loop does just this that it reads messages from an input queue right. So, this is the incoming network queue, right, and parses the message obtains the URL from the message. Reads a file from the disk, which is correspondence to that URL right, and writes the message to an output queue.

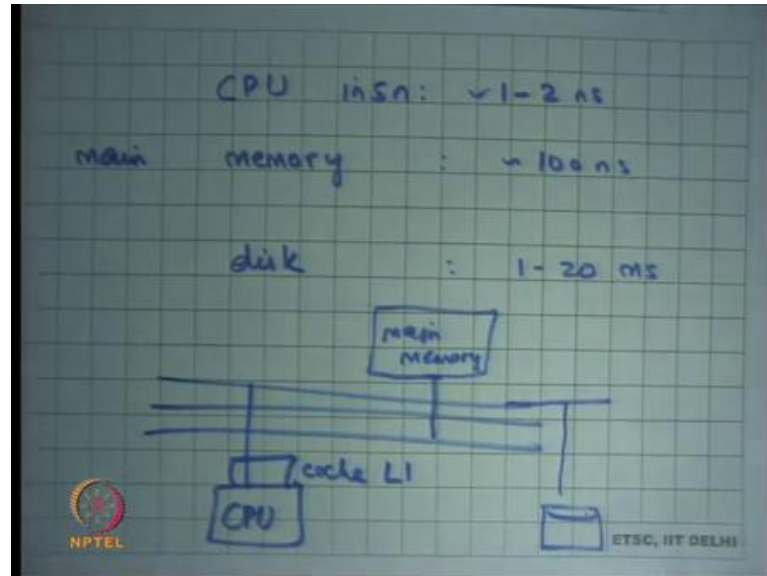
Let us say this is the output queue or outgoing queue, so we just focusing on the server's logic, so this is the server. And you know there is some logic which is basically picking packets off the wire from the network and putting it in the incoming network queue. And then there is some logic which is the network which is picking packet from the outgoing network queue and putting them on the output wire right alright.

And last time we were discussing that in this entire operation you know reading message from an input queue is as fast you just need to copy some data across memory. Parsing the message involves some CPU instructions, reading URL from the disk is expensive because it needs to access the magnetic disk.

And the thing we were discussing last time is that the magnetic disk is a mechanical device as opposed to all these other things which are electronic devices right. And then

you write the reply to the output queue right. So, we said that this one is this step is the slowest of them all and let us look at how slow it is.

(Refer Slide Time: 03:15)



So, we said that the CPU instruction that executes without having to go off chip right. So, when we when I draw this diagram and I say this is the CPU many instructions. And let us say this is the cache this is you know what is called the L 1 cache for a CPU. Let us say and so all this accesses within the L 1 cache and to the registers are on chip access, so they do not have to go outside the chip, so these are also called on chip accesses.

And all the on-chip accesses are really fast, and they because you know assuming you are about one to two giga Hertz processor. They will typically take 1 to 2 nano seconds to execute a single instruction assuming it is not going off chip right. Now, main memory on the other hand is off chip right.

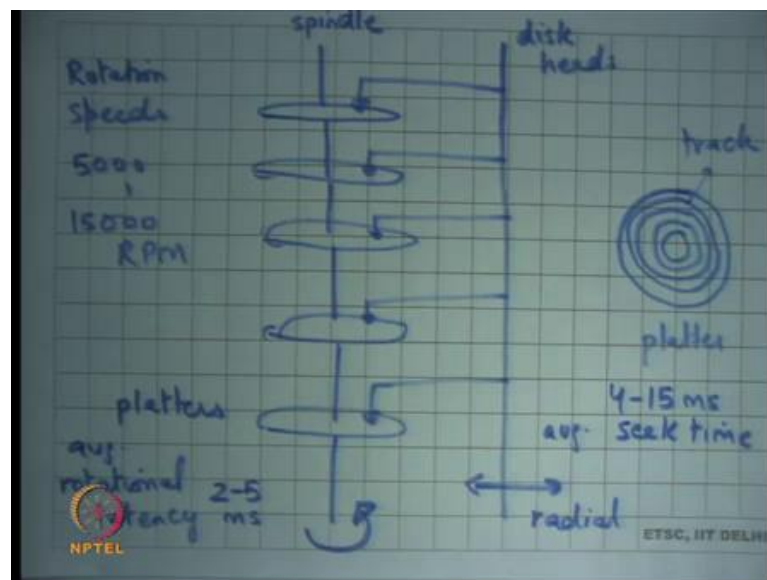
So, this is the main memory and anything which is a cache miss in L 1 has to go to the main memory and it has to go through the bus. It has to travel a longer distance, also it has to talk to the bus to make sure that they are you know multiple devices on this bus example multiple CPUs or other things. Then there has to be some contention protocol that has to be followed, and so, it takes longer.

And it takes roughly 100 nanoseconds to access the main memory. On the other hand, saw something like a disk, the disk transaction also goes through this bus. But,

eventually the bottleneck, so let us say I had drawn a disk somewhere here. The bottleneck is really in the disk device itself because the disk device is a mechanical device.

And we said that the disk usually takes on the order of milliseconds as opposed to nanoseconds. So, actually it is you know a million times slower than your CPU right. So, let us just understand a little bit about how a disk is how magnetic disk works right.

(Refer Slide Time: 05:04)



So, a magnetic disk is organized as a cylinder with such platters alright. So, these are this sort of these circular plates what is called platters all right, this axis is called the spindle. And the spindle is the axis of rotation right, so it just rotates right, and it rotates at a high speed.

And then there is what is called a disk head disk heads actually, so the multiple disk heads. And they are these disk heads that sit on top and are able to read information out of these magnetic platters. It is a bit like the gramophone that you may have seen in old movies or you know in antique sort of things. So, it is a similar thing there is a head which sits on top of this magnetic disk and the disk rotates.

And the head basically reads off information off the magnetic disk and the and basically takes this head is rotating and the to allow the magnetic to allow the disk head to go anywhere the disk head has a capability to move radial right. If I were to draw a platter, a

platter internally is just concentric rings, so this is you know another view of the platter, so if I just look from the at the platter from the top a platter looks something like this, and each of these concentric rings is called a track ok.

And, so the disk head moves radially to position itself on a track. And because the spindle is rotating it will eventually be able to reach the position it wanted to reach, and it will be able to read stuff out of the disk all right ok. So, the average, so both these mechanics, so there are two mechanical movements that need to happen for you to access a disk block. The first movement is a radial movement of the disk head itself. And the second movement is the rotational movement which is usually happening at a continuous speed of the spindle. And both these movements define how long it takes to access a disk block right. So, typical rotation speeds so, rotation speeds of a disk are 5000 to 15000 RPM Revolutions Per Minute. So, if you know if you have a laptop, so some kind of mobile hard disk. Then it will be slower it will do 5000 RPM if it is an inter-enterprise grade hard drive then it will have faster. So, typically ranges between 5000 to 15000 RPM which translates.

So, and so the disk and similarly the time it roughly takes for the radial movement to reach the correct track depends on the distance from the edge to the center of this platter what is you know typical seek times roughly range between 4 to 15 milliseconds that is the average seek time. Once again you know depends on the hard disk and the kind of environment setup in if it is an enterprise environment where you know it will be faster.

So, it will be roughly at the level of 4 milliseconds if it is mobile environment then it will be slower it will be 15 milliseconds and so on. So, let us say I make a request to the disk I want I say I want to access disk block number x. Then it will figure out that x lives on this platter on this track and at this offset.

And, so basically it will first position the disk head at that particular track, and because the disk is rotating eventually it will reach x and as soon as it reaches x it is going to start reading off data from there right. And then once it reads off the data it just puts it in some buffer you know there are multiple levels of buffering that is happening that are happening. One buffer is within the disk device itself, so the disk controller which is maintaining it is doing all this logic.

So, there must be a disk controller that is receiving a request from a computer and then you know implementing all this logic, and that disk controller typically also has a buffer. So, you know all this data that is being read from the magnetic disk is stored in this electronic buffer. You know it is and the buffer has the same technology that you will have for your main memory for example, right. And, so you will basically store it in the main memory and eventually you will send it to the CPU right ok.

So, this is this radial movement, so is also called the seek time right. The time it takes to reach the correct track through the radial movement of the head is also called the seek time of that head. And the time it takes to, so once you are on the head and your from you know once you are on the track the time it takes to actually reach the data you were interested in is called the rotational latency.

So, that is called the rotational latency, rotational latency will depend on you know it is just a matter of chance, if you know as soon as you seek the correct track if it. So, happens that the data you are interested in is right there you will just read it right of it if it. So, happens that it just went by you have to wait for a full revolution before you reach it right.

So, it is just a matter of chance on average the rotational latency will be the time it takes to do half a revolution right. And, so the average rotational latency ranges between 2 to 5 milliseconds alright because that is average. Similarly, I am talking about the average seek time now; once again seek time also depends you know it is also a matter of chance.

You know if you happen if the block you are seeking happens to be very close to your current position then you only have to travel a few small distance and so, seek time will be fast on the other hand if the block happens to be very far then you will have to travel a long distance so, that seek time will be more right.

It is also a matter of optimization often you know as we are going to discuss later in the course. Layout of your file system and layout of your disk should be optimized to minimize the seek time right. Rotational latency you cannot do much about because you know it is just it is a matter of the disk is continuously spinning.

So, you know where you reach when you reach there is sort of completely random. But the seek time you can optimize by making sure that things that are likely to get accessed

together are close to each other. Or the tracks at least either they are on the same track or they are on very close tracks each other alright.

This also means that random accesses to the disk are more expensive than sequential access to the disk right. Let us say I just say I want to access x then y then z you know these are complete and x and y and z are completely random disk numbers. Then it will you know there will be a lot of movement of the disk head to be able to get from x to y and then y to z and so on. On the other hand, if you say I want x 2, x plus 100 that is very fast right, because it is slightly to be on the same track.

So, you just say you know you do not pay the seek time again over and over again you only pay the seek time. For the first block you also pay the rotational latency only for the first block all the other 99 blocks you only pay the data transfer time right which is just now going that much distance and data transfer times are actually much smaller typically than the rotational and seek time. So, random accesses are more cost are costlier than sequential accesses and what is the exact trade off we are going to look at when we are going to study the file systems.

Student: Sir is the movement of disk heads independent of each other or.

Its independent of each other yes, the disk heads and although I would draw it like this, I mean typically you will basically have independent disk, and there is a lot of optimization you know. So, these are the you know what I have drawn is a very strict diagram kind of thing, but there is a lot of optimization that goes on in the middle you know the.

So, and this is all the specialty of people who manufacture these disk devices alright. So, with that we basically you know end up with a number something like 1 to 20 milliseconds to do a disk access right, and where does that leave us in terms of a web server.

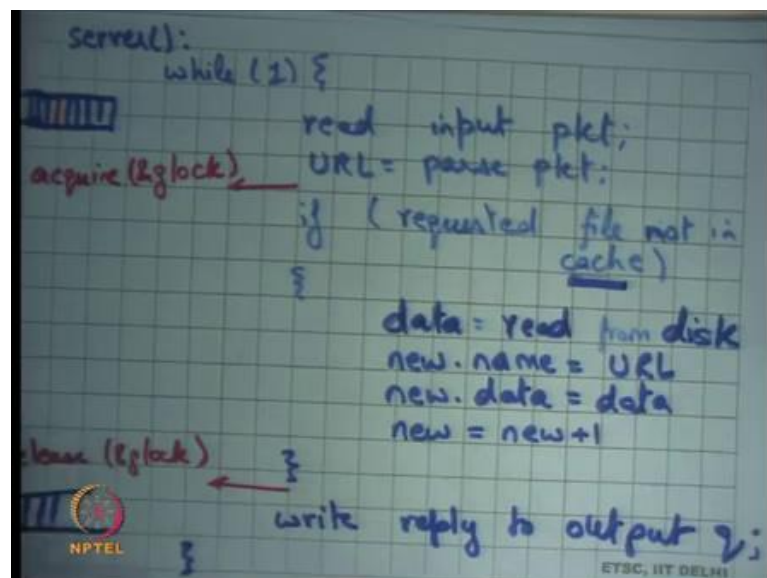
So, in this web server it basically means that every access, every request to the web server will take roughly 1 to 20 milliseconds. Because, you know this is by far the longest slowest step and everything will get serialized here. And you will basically every request will take 1 to 10 20 milliseconds let us say.

And so, how many requests can the web server's service at per second? So, what is the; what is the throughput of the web server 1 upon 10 milliseconds that is roughly 100 requests per second right. So, with this kind of a code you will only be able to service up to 100 requests per second.

So, what will typically happen is let us say you are getting requests which are at a higher rate than 100 seconds. Then they will start they will eventually fill up the incoming queue and packets will start getting dropped at the incoming queue right. And so, you are basically operating at 100 requests per second. 100 requests per second is not a very not a good number at all modern web servers deal with tens of thousands of requests per second alright.

So, what do what is the first optimization that comes to mind, well the first thing is you know why I have to read from the disk every time. It is quite likely that the same pages are being requested by multiple clients which is often the case right. So, once again we are taking the taking this com taking advantage of locality spatial and temporal locality. And we are going to use caching and we are going to cache the contents of the disk in main memory right.

(Refer Slide Time: 15:33)



So, let me rewrite this code for the server and I am going to say while 1. Once again read input packet this short handing it little bit. Let us say URL is equal to parse packet, and then if requested file in cache, then is not in cache is not in cache. Then data is equal to

read from disk, new dot name is equal to URL, new dot data is equal to data alright. And let us say let us have some counter which says new is equal to new plus 1 here all right. And then I just say write reply to output queue right.

So, basically, I have just implemented a memory cache in memory cache, and I will first check if the requested file is not in the cache. Then I will read the data from the disk allocate an entry in the cache and I am sort of assuming that there is a lot of free space in my cache.

So, let us not worry about cache replacement and other things. Let us just say as this point are called new and the new is the current empty slot. So, you just fill up the empty slot with some data and it is called funny URL and you increment the new right. So, this is better because if there are if 90 percent of the requests hit in the cache then 90 percent of the requests can be served from the cache 10 percent of the request need to go to the disk.

However, still if you think about it, you have an input queue right; if even if one of these requests hit misses, so let us say missing one is this red one. So, even if one of these requests misses all the future ones have to wait right. Because, even if one of those misses the thread basically gets busy accessing the disk and waiting on the disk and disk is going to take milliseconds, and so all the future requests are going to wait right.

So, that I mean eventually the throughput of the system does not really improve even if there are a few misses in the system ok. And more importantly the latency the user the perceived latency so, when you; when you; when you type your URL on the browser the user perceived latency becomes very bad. Because anybody who is blocking you know even though you your request could have would have been served very fast. Because there is somebody standing in front of you whose request would take a long time you have to wait for that request to get seek.

Student: We can maintain a new queue just to keep track of just to.

Just to keep track of.

Student: The missed ones (Refer Time: 19:03).

Right, so one way to do this is basically maintain a new queue and basically you say that if requested file not in cache put it in different queue. But then do you need to start the disk and you need to have some thread that is waiting on that disk while you start taking another request right.

So, there is some amount of concurrency that has to happen. Even if you make two queues you also need two threads of control; one thread of control is going to wait on the disk and another thread of control is going to pick up the next packet from the input queue right. And how many threads will you maintain?

So, once again the solution that has been proposed as let us have you know at this point if the requested file is not in the cache. Do not do all this in the context of the current thread. Spawn a new thread right and let that thread wait on the disk and I can go forward and pick up the next request right.

So basically, we are talking about some kind of multithreading and question is how many, so one option is do it you know on demand. So, every time you get something like this spawn a new thread as soon as it finishes you know join it, so that you know terminate it and so, you can you can do it in that way right.

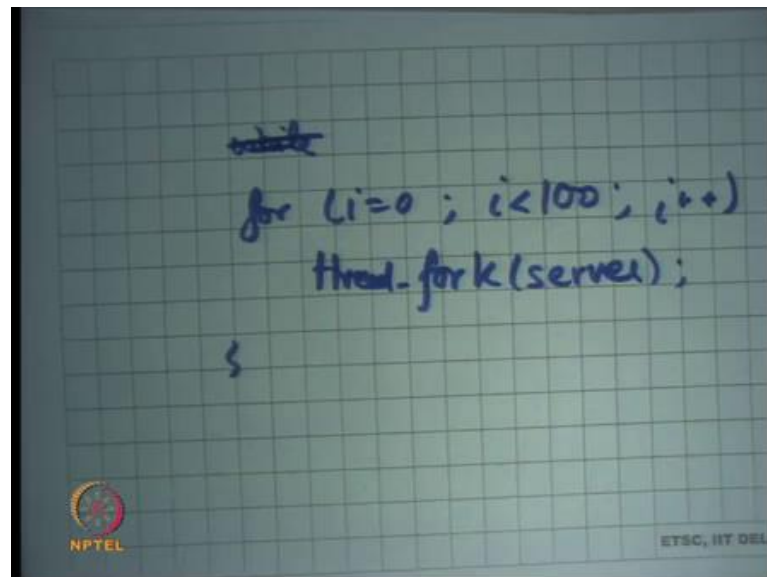
And in either case basically what you are saying is that you need to have multiple threads of control and you need to have concurrency right. So, basically this is an example where you need concurrency to improve your through system utilization and also reduce your response times.

And this is an example where there you need concurrency because there is there are two devices in the system there is a CPU and there is a disk. And you want that both of them should be kept active, and the way to keep both of them active is to have multiple threads one you know few threads is waiting on the disk this is much slower.

So, when many threads can be on the disk simultaneously and let us say one thread waiting on the CPU. Similarly, you know we have seen other examples where the multiple CPU's, so if you want to keep multiple CPU's busy then also you need multiple threads right.

So, in one in say another way you can think of a disk just another processing element, and if you want to use multiple processing elements simultaneously then you need multiple threads all right. So, what are you going to do you are going to say let us say one way to do it is let us call this the server, right?

(Refer Slide Time: 21:28)



So, let us say this is the function, which is the server, and so, you could basically say something like this. You say I am going to spawn a 100, so let us say for i is equals to 0, i is less than hundred i plus plus fork server all right.

So, I forked and let us say this fork is basically a forking a thread or if you prefer let us say thread fork. So, it starts a thread and it basically says that this thread should be running this particular function called server. So, now, there will be going to be 100 threads that are simultaneously executing this function called server.

And now, what will happen is there are 100 threads that are executing the server simultaneously one of them or many of them could be executing on the queue, some of them could be waiting on the disk. And, so you have higher system utilization overall right. How do you choose this number 100? Well if it is too small then you know you can have poor system utilization.

Because, you could have you know you could have multiple threads waiting on the disk simultaneously, and the disk could you know take multiple requests simultaneously. On

the other hand, if it is too large then there is no problem from a concurrency standpoint actually if it is you know if you have infinite threads that is good you know completely logically speaking.

But you will run out of your memory resources to actually manage these threads right each thread takes some memory for example, each thread is the separate stack right. So, threads have some overhead, and so if you have too many threads then you know you are going to soon run out of your memory to manage those threads.

So, there is a tradeoff you basically want the maximum concurrency level without having too much memory over it. And so, you and how and, so you could basically you know one way to do it is just spawn threads on demand. And when you hit your memory limit then stop spawning the threads you know that is one that is another way.

But the all these issues all these decisions basically come in the realm of scheduling, how many threads to spawn which thread to run when etcetera these are basically scheduling issues. And we are going to look at them as when we go talk about it later in the class alright.

So, now I have multiple threads that is accessing this code is there a is that no that is not ok, because there are too many shared resources here right there is a shared so firstly, they are there is a shared input queue multiple threads accessing the same input queue you need synchronization right.

Similarly, there is an output queue here right multiple threads accessing the output queue you need synchronization and none of it is present here and bad things can happen right. Because, it is a shared data multiple threads can be accessing it and they are race conditions and bad things can happen and we know what kind of bad things can happen.

Similarly, this cache itself is shared and so, you know for example, that it can happen that one thread comes in it says is the requested file not in cache. Both of them figure out that it is not in cache and they have duplicate entries of the same file. Worst still it can happen that you know some schedules of this code can cause empty entries in the cache.

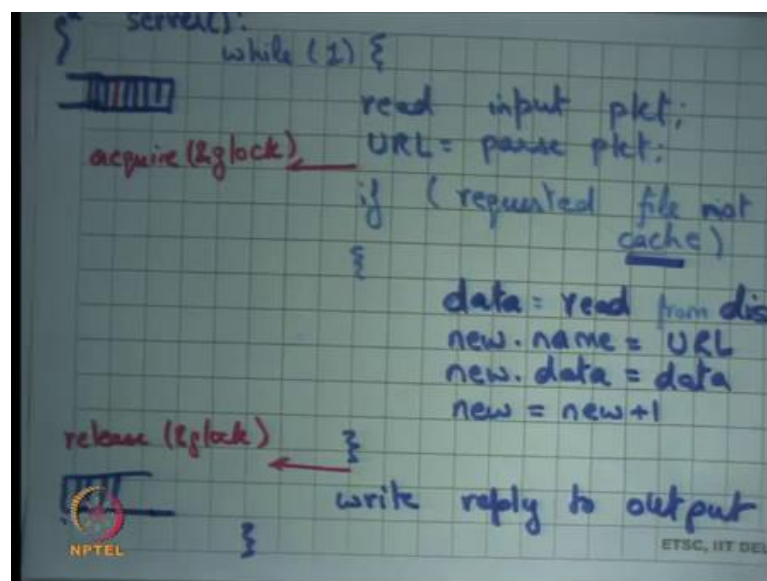
Or some it can even happen worst that it can so happen that you have the URL of one page, and you have the content of another page right all these possibilities right. So, what

do you want? Do you want to use some kind of locks right, you want mutual exclusion basically right and the way we know about how to solve mutual exclusion is locks?

So, what I am going to do is I am going to say let us say; let us say I do and acquire. Let us say I have global lock let us say this lock basically maintains a mutual exclusion for the whole cache and then I have a release lock right. This will take care of concurrent access to the cache; this will not take care of concurrent accesses to the input and output queues.

Let us ignore the input and output queues for some time let us say they are special you know special data structures and we do not talk about how these input and output queues are implemented. But, let us just talk about first the shared concurrent access from the cache. So, you acquire the global lock and you release the global lock is this an acceptable solution have we made any progress from our previous case, so I see some heads nodding, so why not.

(Refer Slide Time: 25:52)



So, what is happening, basically let us say now you have multiple threads ok, multiple threads could be servicing multiple requests simultaneously right. So, each thread is servicing one request, but if there is even one cache miss then you are holding the lock and you are accessing the disk. And so, for that period of time nobody else can take the lock, so what has happened is instead of the request having in the previous case the request was waiting at the input queue.

Now, the request is waiting at the lock acquisition that is the only difference the waiting is still the same. I still once again the problem is if I am standing behind a cache miss, then I will have to wait because the cache miss has taken the lock first and he is going to take a long time. And even though my request would have been would have taken less time. We are basically serialized completely serialized, so this the global lock will definitely not solve the problem.

In fact, you know we have just made things more complex without increasing any performance at all. So, here is an example we would definitely need fine-grained locking to have any benefit of concurrency alright. And so, what kind of fine-grained locking we are we going to have well, what are the shared resources you have these cache pointers new which are pages in the cache.

You have the disk itself is the shared resource right, so when you access the disk. The disk should not be accessed concurrently simultaneously you know accessing with the disk it requires sending some commands to the disk using. Let us say in and out instructions and so, those in and out instructions need to be mutually exclusive.

So, the disk is exclusive the each of these cache entries need to be exclusive, and this check needs to be you know atomic with respect to the other operations here right. And so, you know what the exact solution is basically you know is basically this is a very common scenario where you are maintaining a cache and you need to maintain consistency etcetera.

And we are going to look at this in very a lot of detail when we study file systems. So, file systems have a similar problem you know when you basically accessing your files not each and every request does not go to your disk. It is actually cached in the memory and you have to synchronize between multiple threads making multiple access concurrent accesses to multiple files. And you know here is a sketch of a solution you can basically say I will have a lock for the whole cache here, but then I will try to release a lock.

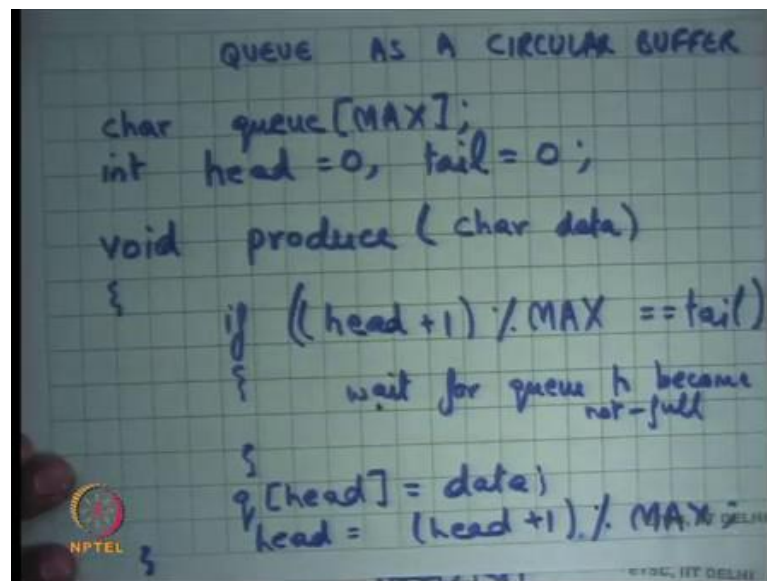
I will only use a lock as soon as I will release a lock as soon as I have done this check, but before I release this lock I will take I will also have a per entry lock. So, I will have a full cache lock and then I will have a per entry lock, and I will you know after I have done the check. And before I release the global lock, I will take the local lock local lock

is a per entry lock and so, this area can execute concurrently for multiple threads and, so you have more concurrency alright.

So, and then you know you have to worry about other things also. So, you have to basically do fine grained locking and there are some ways you can do it and I just sketched one of the ways you can do it and we are going to see this in more detail. And so, you know just to motivate that fine-grained locking issues are actually a little bit complex all right ok. So, now, let us look at let us look at the queues right.

So here is the queue and here is another queue. And once again having queue is a very common pattern in systems in general and in operating systems in particular. And how are queues implemented well you could implement a queue circular buffer right.

(Refer Slide Time: 29:07)



Handwritten code on a grid background titled "QUEUE AS A CIRCULAR BUFFER". The code defines a character array 'queue' of size 'MAX', and two integer pointers 'head' and 'tail' both initialized to 0. A 'produce' function takes a 'char data' and checks if $(head + 1) \% MAX == tail$. If true, it says "wait for queue to become not-full". If false, it sets $queue[head] = data$ and updates $head = (head + 1) \% MAX$. An NPTEL logo is visible in the bottom left corner.

```
QUEUE AS A CIRCULAR BUFFER

char queue[MAX];
int head = 0, tail = 0;

void produce (char data)
{
    if ((head + 1) % MAX == tail)
    {
        wait for queue to become not-full
    }
    queue[head] = data;
    head = (head + 1) % MAX;
}
```

So, a common way of implementing a queue is a [cir/circular] circular buffer right, so let us let us write some code to basically implement a queue. So, firstly, why do I need a queue? I need a queue because there is a network thread that is adding to the queue. And there is a server thread or there are multiple threads which are removing from the queue here. Similarly, there are there is a server thread that is adding to this queue and a network thread that is removing from this queue.

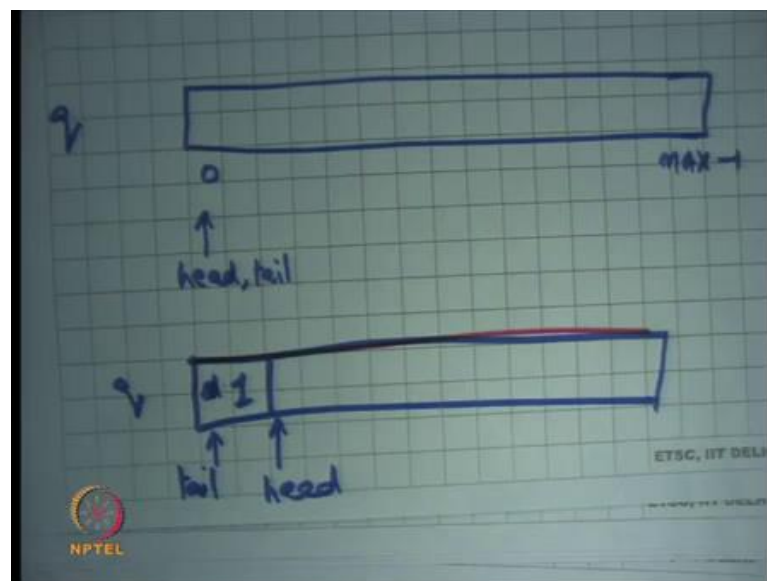
So, the logic is basically, and it is a first in first out queue right a queue basically by definition means it is the FIFO, the First in First Out. So, one thread can add to the queue

and another thread get remove from the queue and you will get a FIFO ordering on that. And let us see how a queue is implemented as a circular buffer I know I am sure you all of you have seen this before.

But I am just going to write the code, so that you know when I write about it in the concurrent code you have more context. So, let us say I have an array of characters which is maintaining my queue, and the maximum size of this queue is max alright. And then I have two pointers inside this array int head initialized to 0, let us say and a tail also initialized to 0 right, all these three variables are global variables alright.

Then you have a function called produce depending on who is the producer you know he is going to call produce input queue network is the producer output queue serve as the producer. In either case you basically say produce let us say some character which is your data and you going to say if alright.

(Refer Slide Time: 31:04)

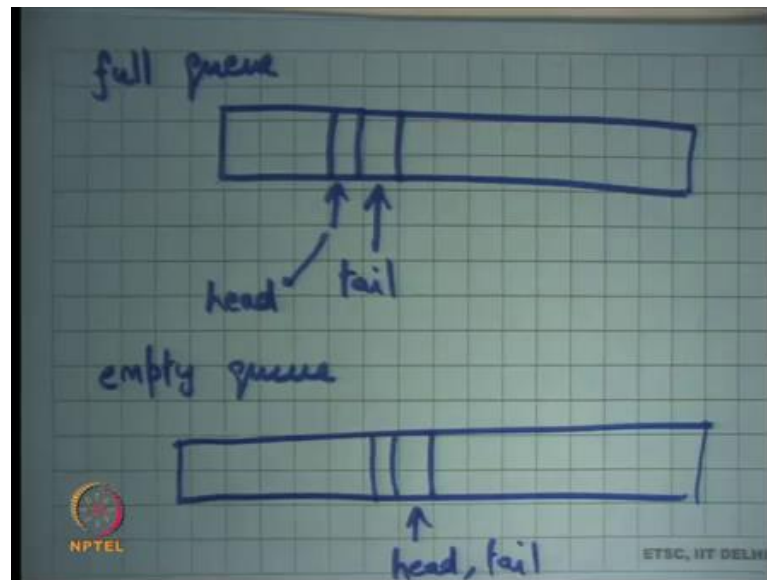


So, let me just first draw the queue just to make sure we understand this. So, let us say this is my array, this is queue right, and this is element 0 and this is element max minus 1 right, initially this is head, and this is also tail alright.

Now what will happen is that when I produce something I am going to increment head so, producing, so when I produce an element in the queue head is going to point head is going to advance and tail is going to be here and this will be the first element right.

Similarly, producer can keep producing and the head will keep incrementing till head becomes equal to tail minus one right in a circular way. So, I just keep incrementing head and, so the consumer may also increment tail and the queue becomes full.

(Refer Slide Time: 32:19)



So, the condition for full a full queue looks something like this tail is here right and head is just behind the tail right. This is a full queue, if I try to add one more data item I cannot do it without having to overwrite something that has not yet been consumed.

So, this is a full queue right, what is an empty queue? What is an empty queue? Tail and head are equal right. So, if head is equal to tail then that is an empty queue, basically the tail has basically hit the head. And now, I cannot move any forward for anything you know I cannot consume anything there is nothing forward ahead that is useful alright. So, what will produce do it will check if the queue is full if it is full then it will let us say return an error.

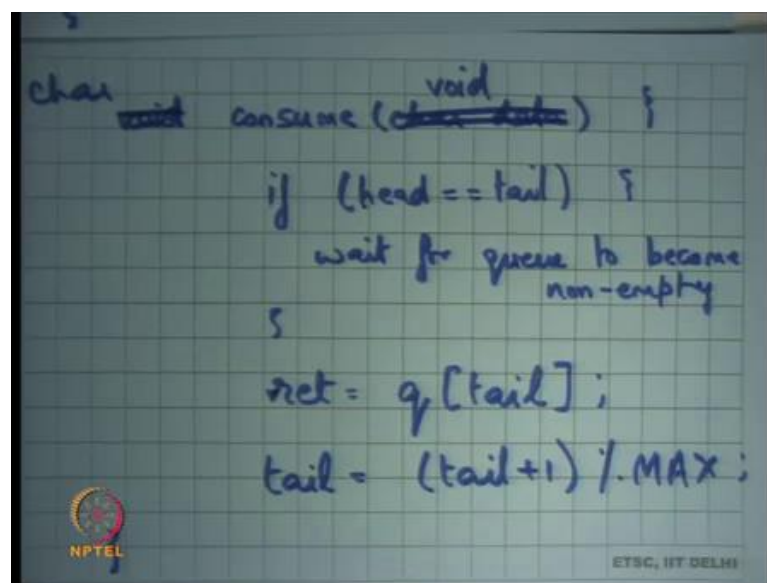
So, it will basically say if head plus 1 and I will do it in a circular way modulo max is equal to tail. Then now let us say error here right let me just leave that as a blank otherwise I just say q_head is equal to data and head is equal to head plus 1 mod max right.

And similarly, you can write the code for consume which I am not going to write here right. So, what should you do here? If you are just operating as a single thread and you

call produce, then you may want to say I want to give an error here right. So, you may say error here you may say error at this point.

But if you are executing in a multiple thread world and you have a shared queue which is shared by multiple people you may want to say let us wait right because there are other threads that are running at the same time. And so, it is likely that even though the queue is full right now it will become empty in some time right. So, one option is wait for queue to become not full right ok.

(Refer Slide Time: 35:00)



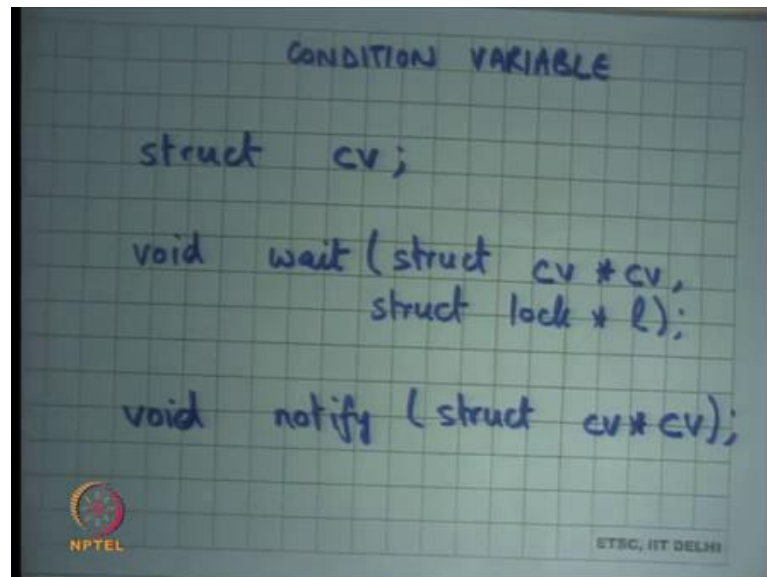
Similarly, in the consume I could have a similar thing here, if head is equal to tail then wait for queue to become nonempty alright. And so, sorry the consume has a void argument and returns a character right you can say return value is equal to q tail right. And tail is equal to tail plus 1 mod max alright ok.

So, now my question is how does, so there are multiple threads one thread is calling consume one thread is calling produce it is possible that the produce finds the queue full and it waits for the queue to become not full. Similarly, it is possible for the consume to become and so, for the queue to become empty and for the for a thread to wait for it to become nonempty. This kind of synchronization this is also solved for synchronization.

So, I am waiting for a condition to become true and only when it becomes true will I be able to move forward right. And the constructs that we have studied on that abstraction

that we have studied so far of locks is not enough to implement this synchronization alright. It is not possible with locks to say that you know I want to wait on a condition locks are only for mutual exclusion; I cannot use a lock to say I want to wait on this condition to happen right ok.

(Refer Slide Time: 37:10)



So, what am I going to do? I am going to define a new abstraction that is called condition variable alright. So, let us understand what this abstraction is just like a lock there is a type called struct CV let us say just like there were struct lock and you can instantiate a lock with this type. Similarly, you have a type called CV and you could have you could instantiate this type and then you have two functions void, wait alright. Struct CV star let us say CV and then and there is another function called notify struct CV star all right.

Actually, the wait has two arguments and the second argument I am not writing right. Now, Before I explained why we need the second argument, but let us just understand the spirit of this abstraction.

(Refer Slide Time: 38:21)

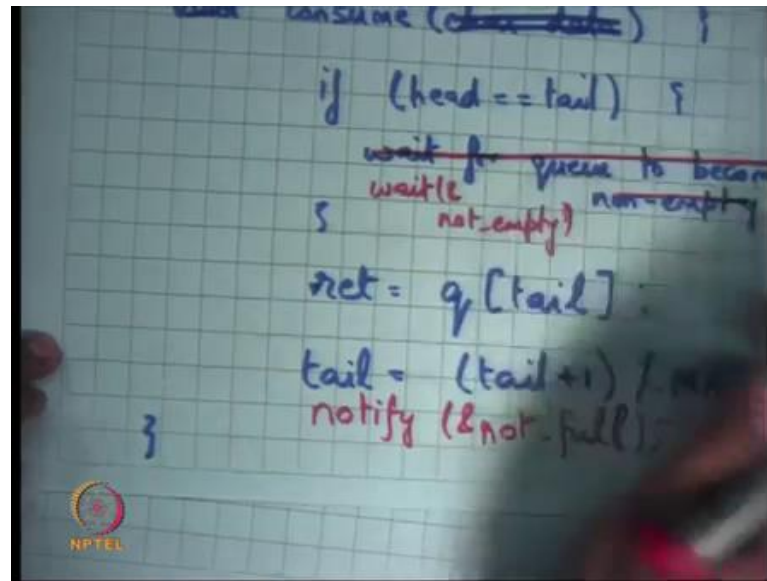
```
int head = 0, tail = 0;
struct cv not_full, not_empty;
void produce (char data)
{
    if ((head + 1) % MAX == tail)
    {
        wait for queue to become
        wait (&not_full); not_full
    }
    q[head] = data;
    head = (head + 1) % MAX;
    notify (&not_empty);
}

void notify (struct
// NPTEL
void consume (char data) {
```

So, the spirit of this abstraction is in the previous example the programmer can define two condition variables and the produce let us say. So, I could define a condition variable called, so that is a struct CV there are two condition variables not full and not empty right. And I could say, so this English sentence wait for queue to become not full.

I can now replace with something; that means, similar thing let us say wait on not full on this condition variable called not full right. So, I have defined a condition variable and basically said now let us say I am going to wait on this condition variable called not full right. Now, how am I going to know whether it has become not full the consumer has to tell me right.

(Refer Slide Time: 39:19)



So, the consumer will basically here say notify not full ok. So, that is how you connect the two basically say that if you find the queue to be full you basically wait on this condition variable you know that I have called that I named not full. And then when a consumer consumes something then clearly after it has consumed something the queue is not full. So, he can say notify not full and the idea is that when he says notify is going to wake up any thread that is waiting on that condition variable right.

So, those the semantics are then when he calls notify if at this point there is any thread that is waiting on this condition variable called not full and get very woken up and it will it will be able to proceed alright. Similarly, you know symmetrically instead of wait for queue to become not empty I could say wait and not empty right and here I could say notify not empty ok.

The effect of notify is that if there is a thread that is waiting on that particular if there a thread that is waiting on that particular condition variable, then it will get woken up it will return from wait. On the other hand, if there is nobody who is waiting then it will have no effect at all right, it is just a no right.

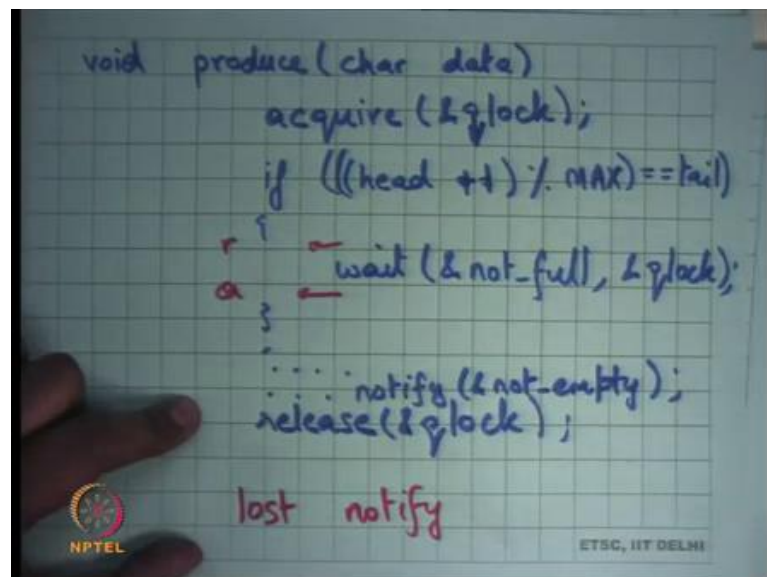
So, it is possible that actually at this point there was no other thread that is waiting on the not empty. In fact, the queue was not empty at this point we are just calling notify unnecessarily it was actually the queue had let us say ten elements and the both the producer and the consumer happily going along. But you are just calling notify

unnecessarily here right, but that is it is not incorrect. It is not it may not be efficient, but it is not necessarily incorrect right, so there are some problems with this code.

Firstly, these are shared variables right q is a shared variable, head is a shared variable, tail is a shared variable. And the two at least two threads one is a producer and one is a consumer and both of them are accessing these shared variables. And they are not doing any sort of mutual exclusion in their accesses and, so bad things can happen right ok.

So firstly, I need I also need locks apart from condition variables I also need locks in this particular code right, because these shared variables that are being accessed. So, not only do I need this kind of wait notify synchronization I also need locks to be able to maintain mutual exclusion on my accesses to these shared variables right ok.

(Refer Slide Time: 42:06)



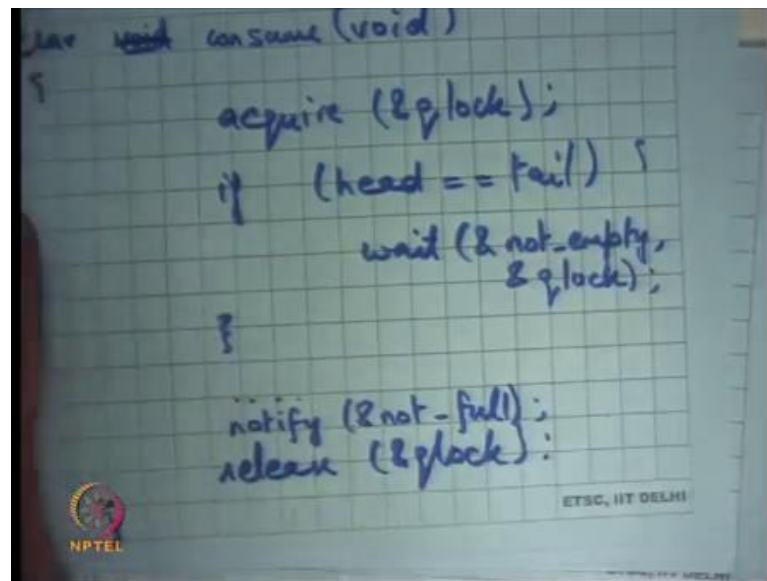
So, let me rewrite this someone say void produce char data and once they acquire. Let us say I have a lock called q lock right, and then I say if head is equal to tail sorry head plus 1 mod max is equal to tail then wait on not full. And I am going to leave it empty because I am going to say something else later. And then I am going to say you know whatever was earlier cause to produce something and then I return.

So, that is my produce code alright and of course, I release the lock here right. Acquire the lock I do some operations if the operations require me to wait, I wait and then I get notified and then I release a lock yes.

Student: Sir, while waiting do we need to just release the lock because we were switching to.

So, the question is while waiting do I need to release the lock yes, so I am going to come to that very soon alright, so good.

(Refer Slide Time: 43:43)



So, let us say similarly void consumed is are let us say char consume a void is basically going to say acquire. It is also going to acquire a q lock it is going to check if head is equal to tail then wait on not empty alright. And then do whatever you like and then release q lock right.

So, now, as was pointed out there is a problem with this code one thread acquires the q lock starts waiting finds the queue full starts waiting on this condition variable called queue not full, but it has not released a lock right. So, if it has not released the lock there is no hope that a consumer will be able to get the lock.

And so, the consumer will get start waiting here, the producer will start waiting here, and you have a deadlock right. So, what did you want to do? You wanted to release the lock before going to wait alright. So, one option is to call release here right, so let us say release now say r here and then do I need to re acquire the lock when I come out of a wait?

Student: Yes.

Yes, because you know I am going to have to execute this, so I need to reacquire, so I call release before wait and call acquire after wait.

Student: Can we make them atomic.

Alright, so you know, so there is a suggestion can we make them atomic. So, the so firstly, this is this sounds this sounds I release the lock then I go to wait then I come out of wait somebody called me notify I come out the first thing I try to do is acquire the lock. And then I get the lock then I can you know I can just go forward.

There is a problem as there was very rightly pointed out that these operations of releasing the lock and actually going to wait are not atomic with respect to each other; let us see what something bad that can happen. Let us say the producer basically comes in finds a queue to be full he checks; his condition finds a queue to be full he releases a lock he releases a lock.

And he is just about to go to wait before he goes to wait consumer comes, in consumes the element signals not full, but that signal will be wasted right. So, that signal will be wasted, and so, the consumer has sent it signal on that notify. So, it will basically say notify that notification will get wasted and then the producer will go to sleep.

So, it is like saying that I have released the lock, consumers can now come in consume the entire queue you know and send signals. But you know I because I have not yet gone to sleep those signals are those notifies do not mean anything to me, then I go to sleep right. And now, I will keep waiting forever the consumers have gone done their job they are for as far as they are concerned, they have notified. The producer was you know, has not started going has not started sleeping at that point and so, you know he is basically.

So, you know one analogy to this is that let us say you know you basically release the lock which means you allowed other people to come into your house, but now you have to go to sleep. So, you other people have come into their house and they have you know they have changed some things and they have also said you know wake up this also triggered the wake up alarm.

But, you know you have not really reached your bedroom and so, you have not started sleeping after everybody has gone then you go to sleep and you will keep sleeping

forever basically you know. So, that is the and so, let us say this is the lost notified problem right that is a bad example, but.

Student: [laughter].

At least I hope it is as this trying to give a physical counter analogy to this thing alright. So, this is the lost notify problem right, so basically the notification is getting lost alright. So, the abstraction is basically the condition variable basically takes two arguments a condition variable and a lock right. And the semantics are that weight goes to sleep on the condition variable and releases the atomic the lock and does this in an atomic fashion right.

So, basically it does all these three things it releases the lock and goes to wait right. So, inside inside this I will basically give this the second argument queue lock. And the semantics are that it will release the queue lock and it will go to wait and it will make sure that nobody can call notify in the middle of these two things right.

It is not possible that when after I release the lock and before I actually went to sleep nobody can call notify, so it is atomic in that sense right. So, you basically going to release the lock and wait, and it is going to be atomic nobody can call notify in the middle of those two operations.

Similarly, the other abstraction is that as soon as it gets notified the first thing is going to try to do is to do a lock acquire right. Lock acquired need not be atomic I mean it is just you know you just going to try or to acquire the lock, and you may have to wait for being able to acquire the lock right.

In fact, this code was not complete, so before you release you basically also say notify not empty alright and here you say notify not full, and here you have to release the lock. So, what will happen is the let us say a producer figures out that the queue is full it goes to wait on not full, but it also releases that lock atomically.

So, if somebody is able to get the lock by that time the producer would have slept definitely right. So, only after it has gone to sleep will the consumer and has released the lock will the consumer be able to acquire the lock it will do all it is operations, and it will signal not full.

When it signals not full it basically looks at all the threads that are waiting on the not full in this case there is only one thread that is waiting on not full and it is going to wake it up. The threads going to get woken up and the first thing it is going to do is going to try to acquire the lock. Is it going to get the lock immediately?

No because you know in this code here the release you know the producer has the consumer has not released the lock yet. So, even though I have notified the producer the producer has just woken up and the next thing is going to try to do is to acquire the lock.

In this code it is possible that this producer the consumer producer gets to run before the consumer actually gets to execute the next statement. And so, it is possible that the lock is still held by the consumer, so he is going to try to acquire the lock he will not get immediately. But eventually he will get it because this one is going to release it, and so now, he can get it and he can continue it is operation right.

So, the semantics of wait and notify that wait takes two arguments the first is a condition variable let me just put this on. So, the first is the condition variable and the second is the lock. Semantics are that the thread which calls wait on these two arguments will go to sleep on CV and release the lock L in an atomic fashion right.

The semantics of notify that it will wake up all the threads that are sleeping on CV alright and that is it, if there are no threads waiting on us sleeping on CV notify will have no effect right. The other semantics of waiter that as soon as you wake up from sleep then the first thing you will try to do is reacquire L right, so alright. So, let us stop here and continue this discussion next time.