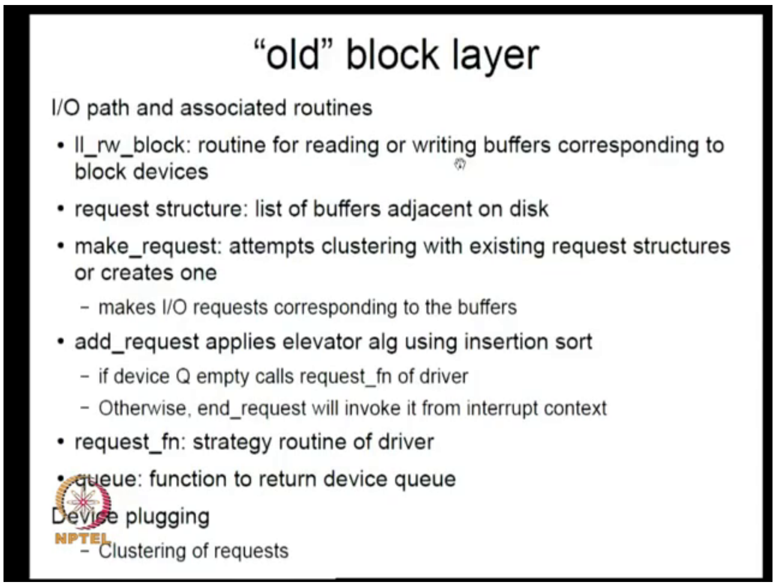
Storage Systems

Dr. K. Gopinath
Department of Computer Science and Engineering
Indian Institute of Science, Bangalore

Storage Interfaces and Device Drivers
Lecture – 19
Device Drivers_Part 3: The design of Block Layer (Issues in Old and New Linux models

(Refer Slide Time: 00:32)



"old" block layer

I/O path and associated routines

- ll_rw_block: routine for reading or writing buffers corresponding to block devices
- request structure: list of buffers adjacent on disk
- make_request: attempts clustering with existing request structures or creates one
  - makes I/O requests corresponding to the buffers
- add_request applies elevator alg using insertion sort
  - if device Q empty calls request_fn of driver
  - Otherwise, end_request will invoke it from interrupt context
- request_fn: strategy routine of driver
- queue: function to return device queue

Device plugging
- Clustering of requests

Welcome again to the NPTEL course on storage systems. In today's class, we will look at some aspects relating to the block layer in some called infrastructure, is available for you to write block device drivers. I think I see, you might have guessed by now, you have a fairly complex system, the only way you contain the complexity is by providing various types of layers, each layer doing some part of the job like for example, when you want to call a use, some specific capability in the operating system, you might use a system call, but you usually do not use a system call directly. What you do? You see first make a call to a life C function, which encapsulate the system call and that life C is some kind of layer which actually takes your arguments and converts it into the form and also does all that code required to move from user space to kernel space.

And finally, the call goes into kernel right, your own basically, the thread, you came in the thread, you are using for making a call right, that is still the same to walk into the kernel, but the arguments have been massaged a bit that mean changed a bit. I think you might have seen in the previous class that you might use a file descriptor, but the time when it comes to the kernel, it is actually a struck file start or something all might come right. You must have remember that we looked at the driver, we look. So, there is some another layers there. So, life C one layer in some sense. Similarly, in the file system in the kernel, if there is a storage service provided, you might be doing it to the file system, the file system might itself be composed. Now, what is called a virtual file system, VFS layer and there on top of it a regular file system like EST 3, EST 4, etcetera.

That in turn we will use what is called a buffer cache; kind of algorithms; we shall do some buffering, that in turn we will use a device driver, specifically return for the device, that in turn will possibly use some infrastructure, that is provided by the kernel. So, that all block devices can use it. This is a block layer and the block layer is the thing which actually finally, talks to the actual device driver.

So, there are multiplicities of layers there and the reason why you do it; is because it is very hard to get any of this thing is working. So, you try to reduce the complexity by giving this layers. So, first I will discuss A, what I will call old block layer that it is previous to, Let us say approximately 2.6 in glance kernel. So, basically, if you look at the thing, the routine looks like the following. So, basically, you have A, if you want to make A read and write request, it first call something called ll rw block.

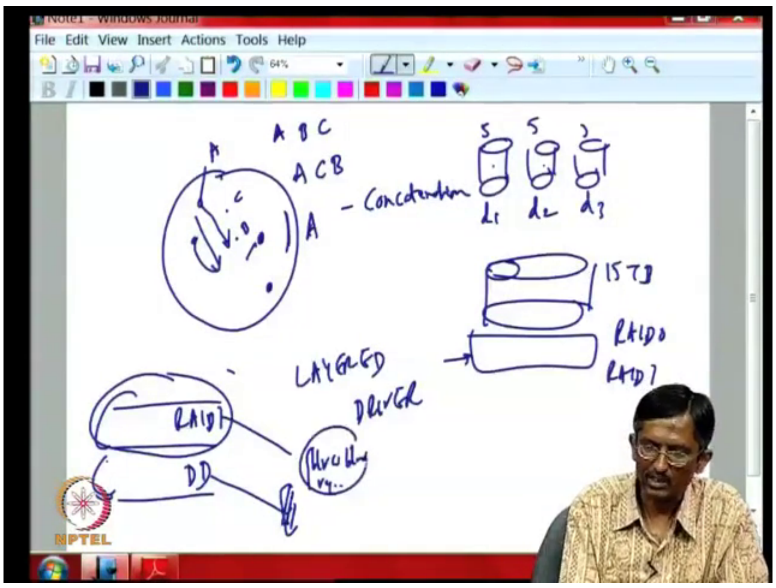
This particular block, by the way, this particular routine is no longer what you might call the advisable for you to use. Now, this is what is called a deprecated routine, there are other routines runs in some, come in a old block layer and it is somewhat simpler format to discuss these thing amuses.

So, when you get a read and write request, you call ll rw block right and basically, there will be, you want to read or write buffers and these buffers are basically the one that is in the buffer cache. So, it also is something called request structure, which is a list of buffers adjacent on disk. I think as we discussed before one of the critical things that you have to do is to cluster things, a disk performs best when all the things that we are looking for a contiguous, all the blocks what you want to read or write.

Then only you get some reasonable performance otherwise you are throughput maybe 1 100th or 1 10th of what the disk can actually do, that is why it is a critical part request structure list of buffers adjacent on disk. Now, if you want to make, if you want to read or write; basically, what it is, you want to take that read or write and see if you can cluster with existing request structures, as you mention what is request structure, is the list of buffers adjacent on disk. So, new request has come in, you want to see if it is somehow adjacent to one of the previously existing requests, that is what this make request us, if it finds that it is somehow contiguous with one existing request structures, it just enlarges that request. If you does not find anything of that kind it creates anymore.

And in the next step is to make the IO request corresponding to these buffers and when you do that it caused something called add request and basically, the idea here is given that you have contiguous writes or reads. What you next do is, you want to see how best they are serviced, you can do. What is called varieties of algorithms for scheduling those requests? For example, you might, if you are having a request for example, on the disk.

(Refer Slide Time: 06:12)



There is a request here, let us say this A and there is A request B here and there is A request C here right. Suppose, the order is in the ABC right, that the way they are coming, what you often do is, you find that you may find it easier to service request in this order first, that is A followed by C followed by B rather than do it from A, then go back to B, then go to C, etcetera. That is much more moment, because it is a mechanical system, electromechanical system and this take time.

So, the best thing is to sort it that is what you going to use. So, we can use what is called elevator algorithm. There are many types of elevator algorithm and that is, it depending on what you have something call C scan etcetera, there are various steps things all. So, and you can typically, what we are discussing right now is, what was there in Linux 2.2, this about almost 10 years back, 10 or 12 years back. So, it basically act it, takes your request and sees in where it should be serviced. Now, thing is now the device, could be right now busy or it may be not busy.

Suppose, it is not busy; that means, no activity is going on a disk, then in the context of the requests that came in, you make a, you call, request fn basically, these are thing which actually does, actual IO, you issue the IO; otherwise what happens is that the disk is already busy and anyway you cannot ask it to anything new right now. So, what you do is you will ensure that when the previous request, because the disk is busy. So, when that disk completes its work, it is going to interrupt and end request is basically, the interrupt handling routine for a disk request. When it is completed, that guy will invoke request function. So, basically what is happening is that, you are requesting an IO in the process context, if you see that the disk is completely idle, nothing is happening there, you came in from your application space, you changed your mode, is still the same person.

Walked inside the kernel and then you see the disk is idle. The disks are idle, then you can actually issue it in your process context whereas, if the disk is busy then you cannot issue another request, because it is anyways acts do something. So, what you do is, you just queue it and then you are done with the job done with your request and because the disk is busy, it will certainly the disk has to finishes some type that work, whatever it was done of course, the disk becomes bad and it never finishes then of course, you have in a strange situation, something is bad with the system. So, the disk has completed the previous work therefore, there will be interrupt therefore, there will be a called back that call back or interrupt handler that guy will notice that disk there are multiple requests sitting out there, I will, it will issue it.

But will be issued in the context of the interrupt handler basically, in the context of a in the, what is call interrupt context? It only done in the process context that something had to keep it back; that means, that the party who is doing; it has more, it has got his own stack, he is complete unrelated to the stack of the party who wanted the service, there is no, there to complete 2 different things in some sense is the proxy, which is sitting there and doing it for you. The proxy is a different person than the party who wanted read and write.

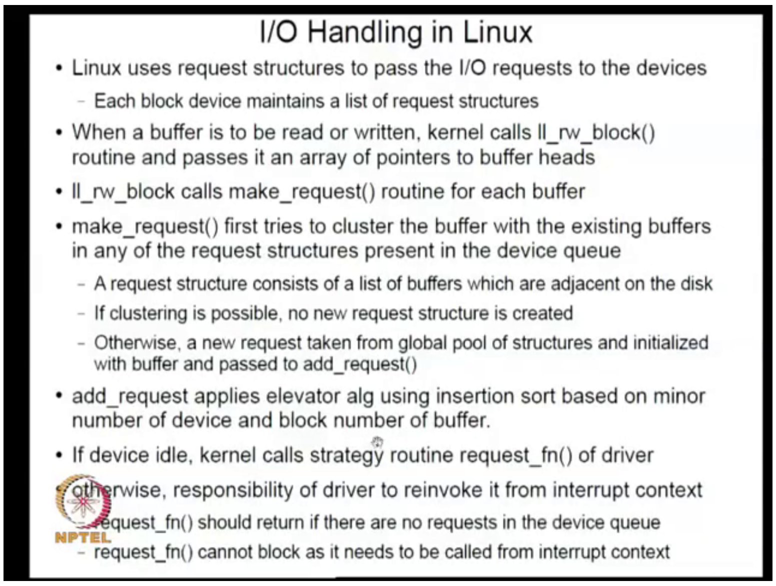
So, that is one part of the thing, the other thing is device plugging basically, we will again go through a bit of this, slightly in more detail device plugging in the sense that this disk performs best. A disk performs best in case you have lots or request here, request then what you can do is you can swap them and then you can use elevator algorithm. So, that as you have to walking like this. You keep request and then every request you keep on doing it one by one right. So, you do not do much of seeking, all the time you are spending, your work, doing work. See this good work, you are wasting time going back and forth.

So, the more number of requests are there it is very likely that you will have reasonable amount of work and less amount of useless work that is just seeking to the get to the next place. So, to make that happen you might also do, what is called plugging device? Plugging when you see that there is nothing, there you plug it. So, that you do not issue the request immediately. The basically, I told you, add request for example, it is going to call the request function of driver all right.

Now, I do not want that happen immediately. I want to wait for some time, I wait for some time and then let us say about 10s of milliseconds. Probably 100 milliseconds, 20 milliseconds, 50 milliseconds. Expectation is that, there will lot more requests coming in and then they will all be captured and then you again sort them do appropriate things and there is some chance that to disk is giving reasonable throughput again. The idea about the device plugging is to do good with respect to throughput, you want to give good throughput rather than quick latency rather than small latency. Basically, the design is optimized for throughput, because what you are doing is even if you have a request, made you waiting for other request to come in; that means, that real it is will increase.

But of the whole system is performing well, it is similar to the idea in multi programming, where when you have lot of users right. Whenever you are doing IO, you go to sleep right, but other parties are working on the CPU, but you will get a chance much later, because all the other professor are could be doing something, you have wait for your turn; that means, your latency will increases, but the system as a whole is being used more effectively. The same idea here also or exact same idea here.

(Refer Slide Time: 13:17)



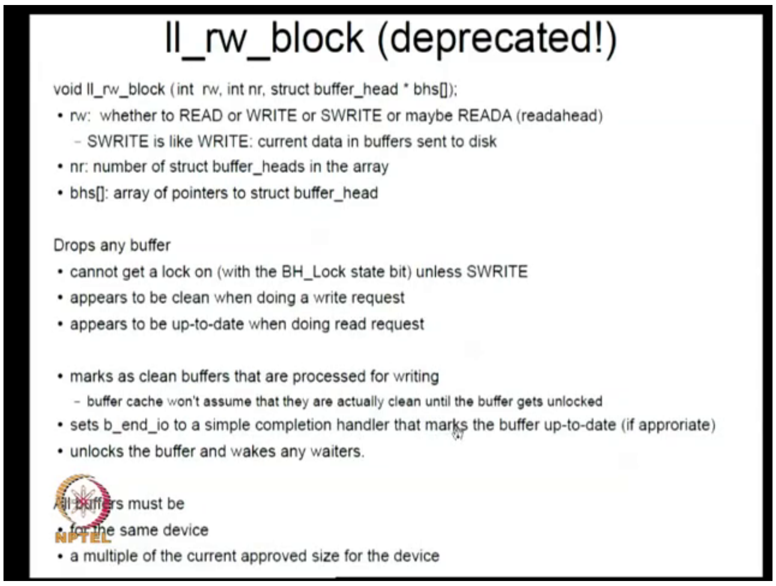
I/O Handling in Linux

- Linux uses request structures to pass the I/O requests to the devices
  - Each block device maintains a list of request structures
- When a buffer is to be read or written, kernel calls ll_rw_block() routine and passes it an array of pointers to buffer heads
- ll_rw_block calls make_request() routine for each buffer
- make_request() first tries to cluster the buffer with the existing buffers in any of the request structures present in the device queue
  - A request structure consists of a list of buffers which are adjacent on the disk
  - If clustering is possible, no new request structure is created
  - Otherwise, a new request taken from global pool of structures and initialized with buffer and passed to add_request()
- add_request applies elevator alg using insertion sort based on minor number of device and block number of buffer.
- If device idle, kernel calls strategy routine request_fn() of driver
- otherwise, responsibility of driver to reinvoke it from interrupt context
  - request_fn() should return if there are no requests in the device queue
  - request_fn() cannot block as it needs to be called from interrupt context

So, again just to look at it one more time basically, you might Linux uses request structures to pass the IO request to devices. The each device will have a request structure.

In older even older device or older designs, there was one single request structure for all multiple disks for example, now the disk for device structures. So, when a buffer is to be read or written, I am just amplifying what we just discussed, kernel calls; this ll rw block and passes it array of pointers to go buffer heads basically, the various buffers in the buffer cache and you can send single. I want to read all these things ll rw block also, can be used to say that I am going to shut down the system. There are some dirty blocks, I am telling it please write everything onto disk because I want to shut down the system, I want everything to be updated, whatever dirty blocks over here, dirty buffers here, I wanted to be attempt. So, ll rw block is caught (Refer Time: 14:26) and ll r rw block, it will make a request routine for each buffer.

(Refer Slide Time: 14:33)



ll_rw_block (deprecated!)

void ll_rw_block (int rw, int nr, struct buffer_head * bhs[]);

- rw: whether to READ or WRITE or SWRITE or maybe READA (readahead)
  - SWRITE is like WRITE: current data in buffers sent to disk
- nr: number of struct buffer_heads in the array
- bhs[]: array of pointers to struct buffer_head

Drops any buffer

- cannot get a lock on (with the BH_Lock state bit) unless SWRITE
- appears to be clean when doing a write request
- appears to be up-to-date when doing read request

- marks as clean buffers that are processed for writing
  - buffer cache won't assume that they are actually clean until the buffer gets unlocked
- sets b_end_io to a simple completion handler that marks the buffer up-to-date (if approriate)
- unlocks the buffer and wakes any waiters.

All buffers must be

- for the same device
- a multiple of the current approved size for the device

If you look at the documentation for example, it is like this ll rw block, it tells you whether read or write, how many buffer heads are there and also the buffers that have to be written out. When the read write can be read write or s write or read ahead read and write are the regular ones s. Write is basically the synchronize, write basically is wanted to do f f sync for example, sync you want to flush all the dirty buffers, because you are shutting down the system or for some reason, you have not to do all those things. So, that is what s write, if there is read ahead, you are telling it specifically that it is for I am expecting that, it will be use, useful if you read the next thing also, but I am not guarantying it. So, in case you are giving it explicit hint, that is a read ahead probably, the down steam like an give higher preference to reach than to read ahead.

Because the read ahead possibly may not use it, you are thinking it is good like it will be use, but it may not happen that way. So, if, that is why if you had a choice between read write versus read ahead, you will probably choose to go with read and write not with read ahead, you can keep it as less prior with less priority. So, again if you think a bit more details, there are lot of which are you use things, now these things actually simple. For example, there are multiple buffers here right, suppose, that buffer is locked for some reason what do you do? Basically, here it says that if you cannot get a lock on it, exclusive lock, this particular routine it is assumes that, it is busy for some other reason.

It will just skip over it you do not try to write it. So, if you are writing, if you are interested in making sure that a particular buffer is written to disk, you may have to do it differently of course. There are 2 differently is handled with s, write in case if you ask for s write, because you want to order, you want to synchronize what is there on memory to disk, because you are shutting on the system, there you have to wait, somebody using the disk buffer, you wait till that guy is finish, then you get a chance to lock it, then you can proceed. So, only when it is s write, actually you wait for it, otherwise you just assume that someday will sometime later you can do it.

So, or it is very spoke there is something called, what is called ordered writes? What is ordered writes? When you are, there are multiple buffers, which have been updated and for correctness you have to send block 1 first followed by block 2, you cannot send block 2 first, already block 1 in that case also, we can possibly use s write, you want to say synchronous, I want to write this first, then I go to the next work. So, if there is a block, which is sitting there, I want it to tell it to wait till that you get the lock. Typically, you choose for handling synchronous, writes especially sync similarly, when there is a buffer which is expected to be written, but it is actually considered to be clean right, that is, there is a up to date copy in the disk. You do not have to process it. So, that solve this kind of simple things.

Again request, read request comes in, it turns out that; that means, you are coming through a different interface, you not come through a buffer, buffer cache interface, because I mentioned to you the multiple layers, you can always come through the buffer cache layer, but the other routines made come from some other instead of the buffer cache layer may come through some other routine, some other interface in that particular thing, if you ask for it to read right, it finds that it is up to date, it is not. It is already present in buffer cache and it is also up to date, then you can also, is keep it and then once this particular thing finishes. It marks as clean, because for example, if you are write, if you have done the write; that means, that the buffer in memory is what is same thing, what is there on the disk therefore, you can mark it has clean now. So, again there issues about locking etcetera, until it becomes unlocked, you do not really write it that way, because when it is, it locking, it means what you not, what exclusive use it may be that you are going to use it again, you want to make dirty it again. So, only when you drop the lock actually, it is going to do it.

There are lot of designs of this kind and I just wanted to give you flavour of it by just looking for only one particular thing and there is also a handler which is invoked when this particular thing finishes completion angular and that marks the buffer up to date and then it unlocks the buffer and wakes and there it could also be that, there are parties for waiting for this particular buffer, you may have to wake them up as I have to mentioned in the previous class, they might be a file which five people are work looking that slash, which is a password in your logging in, he is logging in, all of them have to use it right. So, you try to bring it from disk, but if once it is there in disk as you are reading it, he may also try to login and because he is trying to log in what will happen is that The system will notice that, this is being already getting picked up from the disk.

So, you will be put to sleep and you will be saying that I am waiting for this operation initiated by this party once again, if some waiters also possible for these things and then when you finish, you have to check if it is any waiters are here, read waiters, write waiters, again then you basically read waiters only, only read waiters and then basically, what happens is that you awaken the party, it was waiting for this operation to be complete all that you know. So, this particular functionality is now done in different way. I am just giving you an older from this work, to get an idea about how things will build it. So, ll rw work is what we saw basically that guys, what it is doing it is make requests for each buffer, it first tries to cluster as you discussed before.

Basically, it is checking with it and that is why there are, you have to be careful if there are lots of them, you may want to use more sophisticate algorithms, typically in the beginning. They were using standard simple algorithms, just scan the whole list linearly etcetera, the insertion are gets etcetera, but once you are having a large number of these things, it may be that those kind of approaches will not work out. So, probably might you want to use what to called airier trees, red black trees, these have to stay with algorithms right. You can see some other more, this is turns out to be an important issue, if you are designing parallel file systems, for example, in a parallel file system, what happens is that you could be lots of course, let us say, it is a 4,000 node system that can, I may be writing at different places.

So, multiple course stuff could be handled by 1 IO; IO node and; that means, that there will be. So, many IOs that are coming in from. So, many cores. So, the numbers can be quite large, it can be thousand or hundreds. Well you know that, if you use a very simple, one insertions are they settled write, it may be n squared, there can be a dramatic difference between n square and order of log, in these kind of situations. So, are to be careful.

So, everything you do has a certain connection, the current stable art sometimes the request numbers may be quite small and. So, probably insertion is the best, actually faster, then anything, looks going to be think out to rather in algorithm. We have the more slower, because you have. So, much more generality whereas, some day later some 10s of years later, what seem right, we will have to be modified, all the kind of things keep happening the reason, I am saying it is because we worked with the parallel file system, where this was a problem and it turned out that we are getting very strange results.

So, that system to be figured out that order of n square thing is actually giving all the strange results. We have to fix it before we could make some progress. We thought it was something else, but later we discovered the out of n square things were actually making it look they giving also peculiar results all right. So, I think I already mentioned add request. Basically, make requests place to cluster it and then add request is basically doing elevator algorithm and again if you do the sorting, it has to be based on some things and basically is, basically you have to do it on block number.

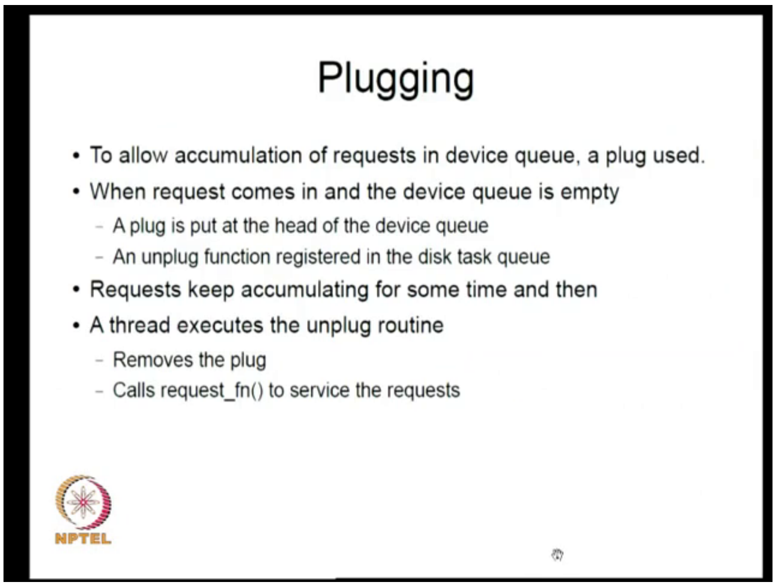
Now, again there are some issues here, it is assuming that the logical block number is same as a physical block number. I think as we mentioned earlier in a previous class right, you had this issue right, then what are called error list? You have a disk and this particular sector becomes back or this sector becomes bad right, then there are something called spare blocks outside here, similar and then the form were actually has got an ability, whenever ask for this, it looks up a small table and say that yes this particular thing is bad. Actually, you should go to describe it on the fly basically, it looks at the logical buffer logical block number last fall and substituted with the actual physical block number; that means, that if we look at this write, you do this elevator algorithm, it is assuming that the block number, logical block number is same as the physical block number.

It will not be the case for a few cases few whenever there is remapping going out, but that is such as ball number that nobody bothers about it. It happens in similar, but if you are a multimedia kind of person, there is an issue basically, because suddenly, you might gets a 20 millisecond gap and the 20 millisecond gap sound samples had to be picked up to every 20 milliseconds. We will finally, see a gap, it signs there, here a click, that is our problem is that ; that means, we have to do some buffering to handle all these things. So, you have to do extra buffering etcetera. So, now once it has happened, you look at the device, I think we discussed already a bit.

The device idle the kernel calls strategy routine request as I mentioned in the block devices instead of read and write, we have a strategy routine and that is the thing actually we does the operation. So, if the device idle; that means, that you came in as a, in the process context and you can issue request function of the thing in the process context otherwise the driver actually has written an interrupt handler, when the IO finishes, it is their responsibility only. Important thing is the request function cannot block as a needs to be called from interrupt context. We already discussed that in the interrupt context, if you block then you can handle the system or it can delay lots of, if you have multiple levels, are interrupts then possibly you will be, but still there are problem.

So, you have to avoid blocking in interrupt contexts.

(Refer Slide Time: 26:00)



Plugging

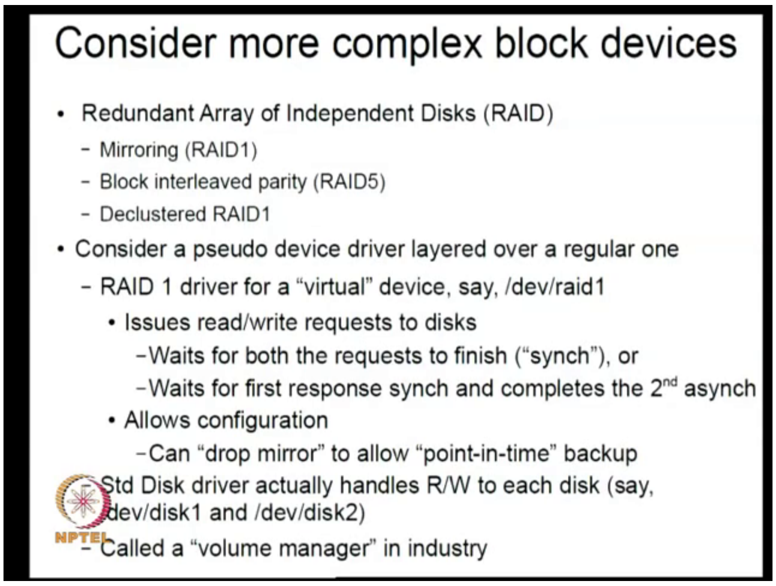
- To allow accumulation of requests in device queue, a plug used.
- When request comes in and the device queue is empty
  - A plug is put at the head of the device queue
  - An unplug function registered in the disk task queue
- Requests keep accumulating for some time and then
- A thread executes the unplug routine
  - Removes the plug
  - Calls request_fn() to service the requests

So, plugging again we discussed it, in some briefly, before basically, we want to allow accumulation requests in device queue and use what is called a plug? Plug is basically a way of calling it. It is basically, way to say that do not process the queue right. Now, that is all it says wait till you get more stuff when request comes in on the device queues, empty the plug, is put on the head of the request queue, it is some kind of a special entry, which is specially marked as saying the plug and then you also register a unplug function, because once it is plugged, there will be some thread, which will be kernel thread, which will be there, which will every. So, often will see if. So, now, plugging has to be done.

As for as stuff has been accumulated; that means, that you have to call a specific routine, which are actually do the unplugging and let the previous stuff gone, because we discussed all this stuff right, all this you handle there has to be of a function that is going to register, which actually does this business of unplugging and calls this part. So, basically if you do this request, keep accumulating for some time and then a thread executes, the unplug routine removes the plug and then calls request fn to service the requests, then basically the way this whole system works. So, the thing important for it remember is that there is this process context, an interrupt context and there could be a system context also because the kernel daemon, which is running. For example, in the case of unplug that will be in the system daemon system context.

Because it is running every. So, often if it is somebody is waking up this particular system daemon kernel, daemon and something like that and that guy is going to execute not in interrupt context, but in system context. So, all three things are possible, it can be doing it in users pay, user context process, context interrupt context or in system context, you have to be clearly, you have been very clear about exactly what these things mean.

(Refer Slide Time: 28:03)



Consider more complex block devices

- Redundant Array of Independent Disks (RAID)
  - Mirroring (RAID1)
  - Block interleaved parity (RAID5)
  - Declustered RAID1
- Consider a pseudo device driver layered over a regular one
  - RAID 1 driver for a "virtual" device, say, /dev/raid1
    - Issues read/write requests to disks
      - Waits for both the requests to finish ("synch"), or
      - Waits for first response synch and completes the 2nd asynch
    - Allows configuration
      - Can "drop mirror" to allow "point-in-time" backup
  - Std Disk driver actually handles R/W to each disk (say, /dev/disk1 and /dev/disk2)
  - Called a "volume manager" in industry

Well that is the way device, a simple block device works, let us look at a slightly more complex block device. Now, you know that the maximum size of a disk is about currently 4 terabytes. So, you want to build a bigger one let us say, I want to build 100 terabyte disk or 5 petabytes.

You cannot get with the single disk; that means, you have to catenate; concatenate multiple disks and then you will get one much bigger logical disk, somebody has to either the operating system knows that there are. So, many disks and then it varies about all of them or somebody is providing it a facility by which it thinks logically in terms of a 100 terabyte disk and somebody is speaking it, from the 4 terabyte design are there. So, basic issue is you will have multiple disk. So, example slash disk, disk 1, disk 2, disk 3. An operating system actually has to know about disk 1, disk 2, disk 3 etcetera and then if we get a request let us figure out where it is and then call that thing right lot of housekeeping work, that the kernel still has to do all file system, has to do and or you provide a big disk.

Which is logically the concatenation of d 1, d 2, d 3. So, this let us say this 5 terabytes is 5 terabytes right. It looks like a 15 terabyte disk and somebody has to fake it and that could be our for example, a new type of device driver, which is designed. So, that it manages all the details for this particular new device, the device is 15 terabytes. You ask for a particular block, it checks whether it is in this first device, second device, the third device and appropriately sends a request to that particular thing, it goes over here or here. So, and that is an example of what is called a RAID 0, driver RAID 0 means simple concatenation that just concatenation or catenation all right, you just join them together.

Now, that is one there is other possibilities also suppose, you are concerned about reliability, you are concerned that the disk may be falling apart or it has got what is called sector levels and you may lose some information. So, you can do what is called RAID 1. What is this? I think the discussed briefly, before RAID 1 is a one keeping multiple copies of the blocks 2 copies typically. So, that if one copy goes bad, you can pick it up from the other block. Now, there are somebody has to manage all these things also. So, you can write a driver, which is layered on top of the regular disk driver. So, you can call, you can devise, what is called a layered driver? So, you have a RAID 1 or RAID 0 driver, which is sitting on top of a regular disk driver.

So, that is basically the; so, let us that us basically called mirroring. Mirroring is what RAID 0 is just simple concatenation, that also requires some extra help, RAID 1 actually, it has copying also they are also requires, some is to help. So, all this extra help can be put into one device driver, a new device driver, which works with a new device, new. So, previously for example, you might have essentially, what we are talking about is we want to device, what is called a pseudo device driver and actually is why we call it, pseudo device driver, because there is no real physical device, new physical device is not there, all you are done is take the physical devices and then give it a logical grouping of it that saw later.

More new physical devices here and we will give it also a new device name. Let us call it slash dev RAID 1. What does it mean from the point of the kernel, there is slash dev RAID 1 and it is basically, composed of some multiple devices which I given all you know about it, thinks of it as a regular blocked device and it says read and write etcetera, but behind the back of this kernel somebody, the pseudo device driver is keeping 2 copies of things. We call it as pseudo device driver, because there is no real physical device there. It is all in memory everything whatever we are doing. It is a, it is some piece of code sitting in memory, which is doing intercepting the reads and writes and then it finally, issues the reads and writes to the various disks, that comprise the pseudo device level pseudo device.

So, you might have a RAID 1 driver, for a virtual device. Let us call it slash dev RAID 1 and it issues this particular driver, issues read and write requests to disks. Now, since there are 2, let us say there are 2 disks, there you can wait for both the request to finish, that is you wait, you issue the request, the first one, then in parallel you can give to second one and then wait for both of them to finish or you can give to one first, wait for to finish, then give the second one, wait for it to finish, there also is possible, which is a basically, if you cannot handle too much concurrency that is a way to do it, but here going to handle concurrency, you pump it to both the disks and then you wait for the guy, who is for both of them to finish or the other thing is possible. This one more design possible, you wait for the first guy, because you know that you are at least putting onto one disk and you say that second one.

It is taking for some reason slightly longer time. I do not know, you wait for a second guy, waiting to be done asynchronous here and get only for the first guy, there also it is possible or you can even do the other things that is, if you see that there is a lot of requests we wanted. What is called load balancing? It may be that there is a lot of requests coming in on 1 disk compete other one right and then you look at the statistics and say that there is a queue size of this much, on this particular disk 1 versus disk 2 and I will the request, will be sort of apportioned equally to both of them, I will decide whom to give it to, because writes have to go to both, where reads can go on only one, because it is a basically, essentially you have copy right.

So, reads can be steered to whichever one is less, busy all that intelligence can be there in your pseudo device driver and this pseudo device driver actually will call the standard device, standard disk driver and what we discussed before we call that guy and that could be slash dev disk, one slash dev disk 2 or whatever in addition to this, the driver, the raid one driver for example, should allow for, what is called configuration or reconfiguration? What is the, let us give an example often times, how do you take a backup? Suppose, you are a very busy, website amazon or eBay or now, they are storing some critical information about various things right and it has to be running continuously, online 24 hours right.

As far as you know eBay goes down once in a while down, in a particular year it does not go, this know something like a time in the night, when it actually stops working as by the sender, because it is market, is worldwide they cannot afford to stop any time, but we also all take back up. How do you take a backup of things, which are when continue activities continuously going on one way did this, what is called drop in the mirror? What is the mean, you have a regular RAID 1, you issue a configuration request, saying that I am going to take a backup, I want what is called a point in time by backup for the particular time, instant, exact time. Let us say t equal to mid night, exact midnight, mid night. I do not care whether it is consistent backup.

Consistent backup means there are some transactions whatever blocks that were affected by the previous transaction should all be sitting in disk, but whatever I do not want situation, where this is a transaction. Some parts of it are not disks, some parts of it are not memory, is take when I made consistent. What you all mean, I am interested in the situation where there is a transaction, it has got some blocks that have, that has become dirty, because the transaction, every single block of that transaction either makes at a disk or no other make it that will be consistent. Now, let us say that that is a desirable thing and that is what applications. Upper level applications like data basis will take care of, but let us say I want to, I do not have information of about that part. What is consistent? What is not consistent, because it is upper level semantics? Now, I am somewhere down here and I want to able to say that I want to take a point in time back up. What does it mean?

It means at this point, in time exact mid time, midnight, without worrying about consistency of the blocks has seem by upper layers. Can it take an exact back up? So, what I will do is, I will drop a mirror exactly at midnight; that means, that I will pretend as if one discuss field or it is taken offline that is called drop in mirror, take it offline and then the system knows now that there is only single disk and then it will read and write only single disk. So, now, we have a disk, which is completely offline. Now, you can copy it exactly, put on the tape, it may take on hour or whatever, because if it is a big disk, 4 terabyte, this reading and writing will take 1 or 2 hours. I moved. So, multiple hours.

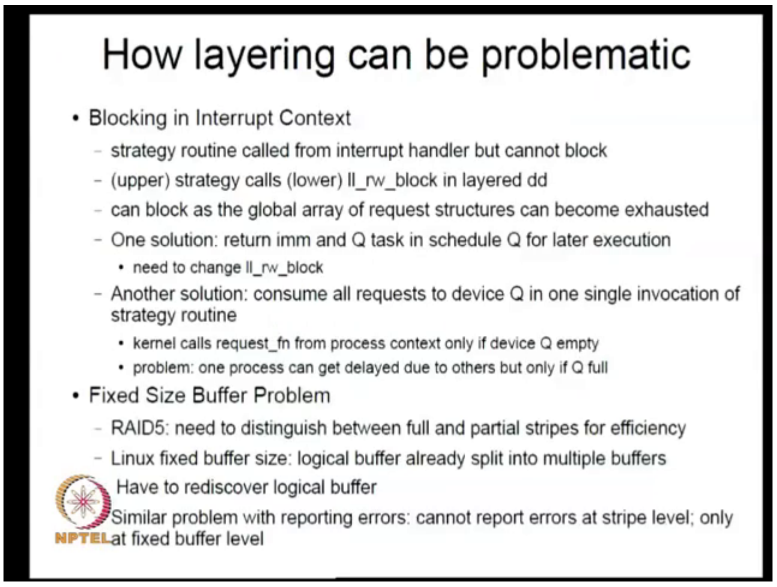
Now, once it is done we want to put it back on to the system, because you want again get the mirroring operation going on. So, basically, what you are doing? You are exposing yourself to some amount of time, when those transactions during that period. You do not have 2 copies. It is a bad thing, but is better than stopping everything, stop in the hole eBay or whatever you doing right, amazon right. Now, some of whole thing. So, you are slightly unprotected for some period of time, but once you have copied it, you want to get it on that. Now, these 2 things will be slightly out of whack, they will not be identical information is there, then you can run a given, the background which sinks it, this systematically goes through it and then sees whatever is a different and then copies it from that disk, which has had all the updates puts it onto the second disk.

So, due to the background till both of them get recent, it is what is called, it is often called re silvering. So, mirror you have a mirror slight, there was got a silver. So, that the back and that mirror is gone, because you dropped it and then you re silver it. So, there is a mirror back ability it is called re silvering. So, this is some other terminologies used. So, you can drop a mirror to allow point in time back. So, basically, for all these things you need a way to talk to the device, what device slash dev RAID 1. So, you should, they will be send IO CTL calls, I mentioned, there are IO CTL calls, which are basically, way to tell the system to do some configuration. I mentioned that in the file systems, you have different CTLS is to configure the file, the way the file actually behaves.

Similarly, you can also have IO CTL also which will tell you how the devices should be here, here what we doing were telling a device for dropping in mirror were telling the guy at this point in time. I wanted to behave like a single disk, not like a RAID 1 device again, when the thing comes back on a; tell the guy. Now, from the, onwards you are going to be a RAID 1 device and please go ahead and start doing the update that happened a 1 disk, please copy it onto the second disk also. One is basically configuration. So, you have to provide in addition to the standard driver, you need to provide additional capabilities by which you can reconfigure the device, first things are called a volume manager. Now, you also have what is called RAID 5.

We also have what is called declustered RAID 1, when we have multiple disks, you can have d RAID 1 across them. So, I am not going to go into details about these things, there is RAID 5, RAID 1 etcetera, that becomes possible.

(Refer Slide Time: 42:19)



How layering can be problematic

- Blocking in Interrupt Context
  - strategy routine called from interrupt handler but cannot block
  - (upper) strategy calls (lower) ll_rw_block in layered dd
  - can block as the global array of request structures can become exhausted
  - One solution: return imm and Q task in schedule Q for later execution
    - need to change ll_rw_block
  - Another solution: consume all requests to device Q in one single invocation of strategy routine
    - kernel calls request_fn from process context only if device Q empty
    - problem: one process can get delayed due to others but only if Q full
- Fixed Size Buffer Problem
  - RAID5: need to distinguish between full and partial stripes for efficiency
  - Linux fixed buffer size: logical buffer already split into multiple buffers
    - Have to rediscover logical buffer
  - Similar problem with reporting errors: cannot report errors at stripe level; only at fixed buffer level

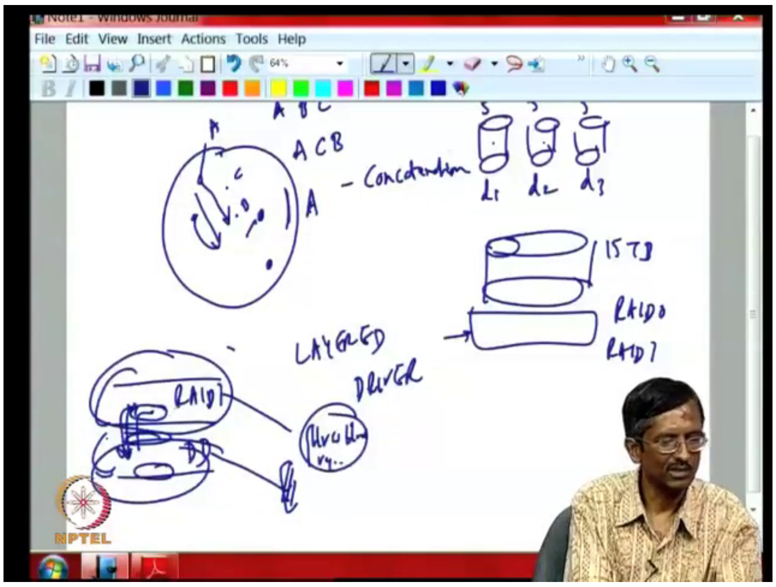
Now, let us think about how this? So, we are going to be some layering, it turns on layering can be problematic. We will just understand, what the problem is. So, I think once we already mentioned that there is a problem with, we said request function cannot block as it is called from interrupt context. Now, a strategy routine called from interrupt handler, but cannot block, I think already seeing the same thing, but what is the, what is going to happen is that.

Since, it is layered you have upper layer and a lower layer. So, it is something like logically when think of it this is the RAID 1 driver, it is the regular, it is driver may be now, these are all and viewing, this also block driver. This also is a block driver, these a regular block driver, but I am using this also as a block driver, because this also there is it is called the same function, because in during also in terms of blocks logical blocks as understood by RAID 1, RAID 1 also in talks, in terms of blocks. So, is this, also is a block driver. Now, this guy has got a request function, it also has is got a LL RW block. So, this has also is got LL RW block.

This guy also has got a request function etcetera, it has also got a make request, it has got also a request everything is there. These guy also has the same thing, these are standard ones that came with the Linux kernel. This is what I was write, I am writing, there is got the same functionality and because of this what is going to happen the upper strategy is essentially called the lower LL RW block. Now, this can the LL RW block can block as a global array of request structures, can become why? Because you notice that if I have a top level RAID 1 or RAID 5 right. One request can involve 5 disk request, actual disk requests for example, I have something like instead of making Raid 1, which is exactly 2 for every logical block, there are 2 copies in RAID 1, it could be 2 copies. It could be the 3 copies, it have 5 copies right, logically I am talking about asking it to do a block write right, a single block, but actually turns out to be 5 of them, because I have to make 5 copies of writes.

In the lower row. So, it turns out this LL RW block it. It has a set of request structures it is going to use and then because I am having multiplicity of buffers, that have to be are involved right, it may turn out that it might turn out of space and what is my problem? My problem is that, because this, the request function right, I am going to call request function here.

(Refer Slide Time: 45:47)



This request function is going to call something below here, and this should not block. The understanding is that, the request function is not block, but since its calling lower guys, and LL RW block, actually block this request function which is being called an interrupt, context can also block which is a problem all right. So, that makes that you need to find some other solution for it, you have to, want to going, which this call request function does not block.

This request function already people have taken care of the clever guys; the Linux Torvalds, all that you will get taken care of it right. Now you are writing this. This request function is going to call this lower level routines right, and if this LL RW thing blocks yours wrong, because the idea is that in the block, device provides you are not supposed to request function, is called interrupt context, because the thing is, when you finish all these writes, each of the writes right. Then you call back the corresponding request function. The completion routine for the RAID 1, because the RAID 1 itself, might need some completion routine which will fix up everything once, because the whole write is complete right and that will be call in a interrupt context, and because of that there could be a problem serious problem, because of the blocks.

So, there are various ways to do it. One solution is to, if it basically can make it sleep made; that means, that you have to change LL RW block of the mainline kernel, which is a, which is not a usually good thing to do. There are other solutions consumer requests to device queue in one single invocation of strategy routine. You all the time make sure that you are always giving the request in process context if are in process context, you can always block, this is no problem, only interrupt context you have problem. So, what you can do is, to ensure that any time you work in, want to look at the queue which is always empty then you know that you are going to do it in process context, if you are able to make it happen, do that way.

There is a problem with it also, because you are now delaying, you are waiting, you are essentially doing other parties, because you have to finish your thing, complete before the other take anyone getting. There is a, there could be some problem here a, even better solution is to pre allocate, buffers it, basically why is it failing, because you are allocating request structures and you are running out of memory; that is why it can block. So, if you want avoid this system, avoid this particular problem, what you can do is. Before you start doing anything you pre allocate all these buffers; that is you are in the upper level layer. You basically pre allocate all the buffers you know, basically it take the worst case situations and then if it so happens that.

You are unable to get the buffers you block, before you get into the interrupt context in the first places and even had various things. So, that also is possible. So, there have to solutions possible, that something you have to think about carefully, when I am writing these kind of drivers. Most important things they use to see, where could they be blocking points, and if it is being called from the interrupts context, you have to ensure there is no blocking going on. And how do you avoid blocking. Typical easiest ways to its, you find out why it is, why it could be blocking if its. So, memory location, then you allocate pre allocate the buffers before you actually do anything. Now at least you have to take as a worst case situation.

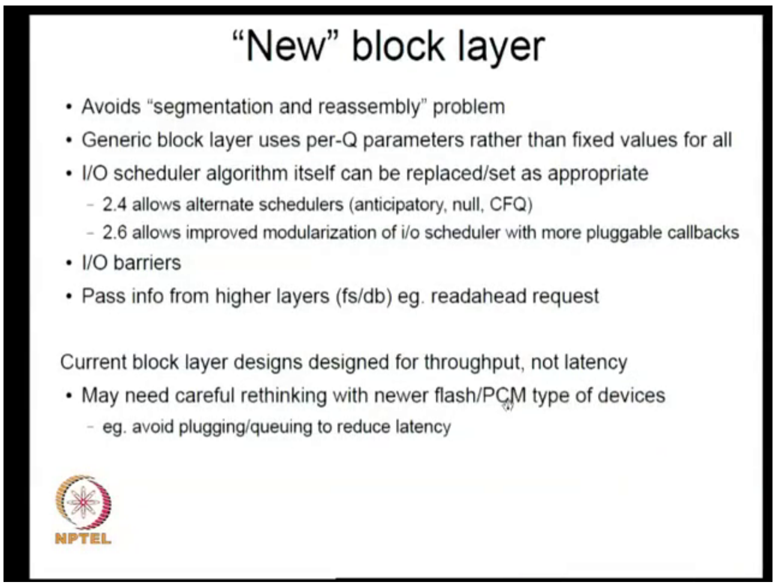
That means you might allocate much more than what is really required, because you might have branches in the code, and then you have to irrespective of which branch you take, you have to find the maximum number of allocation that you can do and allocate all of them before you can walk in. So, that will available, there in case you cannot allocate. You just basically hang that to allocate it in delays things, but you see. So, this is one example. How layering can be problematic, because you are recursively using the same structure at every level, and in a single one level structure, somebody is already taken the trouble of, figure out how to make request function not block, but your request function in turn depends on lower level function which could block.

So, you have to figure out some solution, simplest is instead of either of these 2 solutions simplest is to pre allocate; that is often the strategy units work, the other kinds of problems. There are many problems that layering, actually can introduce that I am talking about to a problem fixed size problem. So, now, if you have, there are some types of devices like RAID 5, I think we discussed already what is RAID 5. It basically have parity right. You take the parity of multiple disk blocks, and complete the parity and store it in the fifth disk or sixth disks or a seventh disk. And if you are reading a writing to RAID 5, you have to do, you have to avoid what is called read modify write cycles, read modify write cycles and.

So, you have to be able to write in full strips, because if you write only partial; that means, you have to read the full stripe and then, only then you can update it, then write it back. You have to read it first, update it then write it back so, but what is happening right now is, that since you, if you 1 x has the fixed buffer sizes, this is all again including this about the issue about some years back, the logical buffer already split into multiple buffers, and then when you finally, ask it to avoid this read modify, a check of it. You have to check whether already is splitter buffers are actually contiguous, and actually constitute, why full straight, write somewhere, I should do the back mapping again. So, again there are. For example, the upper level was slow, if my RAID 1 write finished or not, but the lower level guy will only give you what happened with the disk one failed the disk 2 failed.

Somebody has to again map the failure that disk one, and disk 2 is saying that your RAID 1 write did not succeed, somebody has to doing the mapping also, the various issue of this kind.

(Refer Slide Time: 52:28)



"New" block layer

- Avoids "segmentation and reassembly" problem
- Generic block layer uses per-Q parameters rather than fixed values for all
- I/O scheduler algorithm itself can be replaced/set as appropriate
  - 2.4 allows alternate schedulers (anticipatory, null, CFQ)
  - 2.6 allows improved modularization of i/o scheduler with more pluggable callbacks
- I/O barriers
- Pass info from higher layers (fs/db) eg. readahead request

Current block layer designs designed for throughput, not latency

- May need careful rethinking with newer flash/PCM type of devices
  - eg. avoid plugging/queuing to reduce latency

So, that is as old stuff. There is a new block layer. Again this new block layer keeps changing every so many years. I am just giving you some information about as you take this write, in the recent past it may not be exactly as you take this write now. So, I mentioned there is a problem about this full stripe writes right. You have to look at the structure, because you came from the top full stripe write, you split into small things, but finally, when you issue the writes, you may want to discover the fact that it is a full stripe write right.

Now, these kind of problems have been solved. There are other issues that have been solved in the newer models. For example, there is a lot of flexibility in changing the schedule algorithm, I mentioned previously, considering elevator algorithm. Now you can use something called anticipatory null and completely fair queuing some models. You can change all these things and basically it also allows for input modularization with more pluggable call backs. Basically what it means is, that things like make request, add requests they are made it. You can define all these things yourself and they have made it.

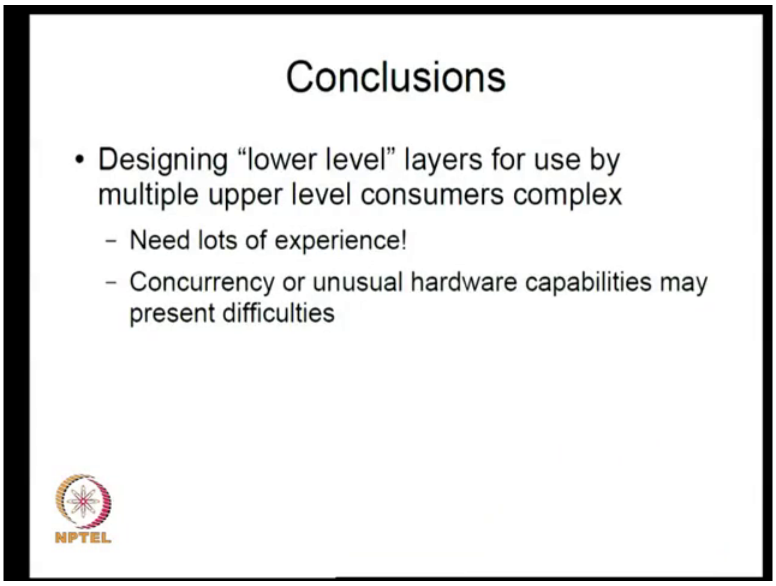
So, that you can put the varieties you want. You do not want to have a standardized; think that only, that all you have to use the other things like IO barriers basically.

We want to say that up to this point, I want all the IOS to be completed before you think of starting the new one. The IO barriers. Similarly also there are issues like if there is information from the higher level available to lower level, it helps lot of times for example,. So, the higher level knows what, its doing what. It is doing a read ahead request. Read ahead means what I am reading at this point. I expect it to continue a sequential read, then if it is tag, does that is read ahead. When it comes down it can see and say there are critical operations here. This is a read write and read ahead. It can decide the total read ahead may or may not be used. So, I will prefer read slash write to over read ahead, that information is coming down if you do not do that tag, tagged information, saying its read ahead.

Then bottom level layers cannot figure out what to do with it. They think all of them are equally important. So, basically it turns out that the current block layer designs are designed for throughput, but not latency. So, when you have things like flash or newer types of devices, not like there, they are not looking like tapes or disks for example, we have to. We are concerned about latency for example, most of this something called PCM. PCM is a new type of memory; that is coming out, but persisting memory that is coming up, and this type of, it actually looks more like memory, regular memory only, is persistent and its quite fast also. And it may also have, instead of reading and writing in chunks of 32 kilobytes.

You can, its persistent memory which is written and read in 16 bytes or 8 byte per 1, is more like a cache way. So, for this kind of stuff latency, you are essentially coming close to daemon kind of speeds. And since it is readable and writeable at byte level, or multiple small byte level. Look at 2 byte a 4 byte level right. You can use it like memory, and therefore, you are concerned about latency at that point, you do not want to, because they are fast also. If you tried doing this plugging business write, you are waiting for a lot of requests to come before the can issued right, then latency become high. So, you have to completely throw away all this stuff or you have to find a special new path, by which you can avoid these kind of things.

(Refer Slide Time: 56:19)



Conclusions

- Designing "lower level" layers for use by multiple upper level consumers complex
  - Need lots of experience!
  - Concurrency or unusual hardware capabilities may present difficulties

So, basically it turns out, if you are designing any of these block layers which are used a multiple block device drivers right.

You want to provide some infrastructure, and what other people are using it on top right. Then it becomes lot of it. It turns out to be fairly complex to figure out how to do it, and it requires a lot of experience, basically the concurrency. All typically one to exploit some hardware capabilities, all these things, it is the initial hundred capabilities then the previous generation of block layer design, if I am not taking this into account right. They are actually have to be again reworked again, and need not have to change their layers again completely. So, there is something that keeps happening every so, often. So, it is always in a dynamic state of affairs, it is not going to be static.

So, I am really sure that currently 3.0 minus 3.7 or 3.8 has sums block layer, which probably relooked that. And probably people who say that flash is going to be the most important thing or PCM is going to most important thing, and we need to have completely some block layer, which is better at it rather than what we are currently using, which is closer to a disk level kind of node also. I think I will stop here.