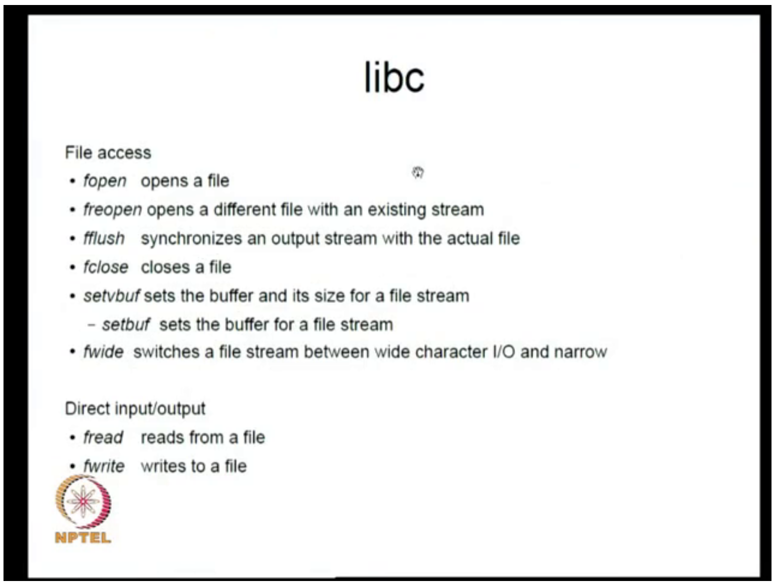
Storage Systems

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

Storage Interfaces and Device Drivers
Lecture - 16
Interfaces to a Storage system continued, Introduction to Device Drivers

Welcome again to the NPTEL course on Storage Systems.

(Refer Slide Time: 00:24)



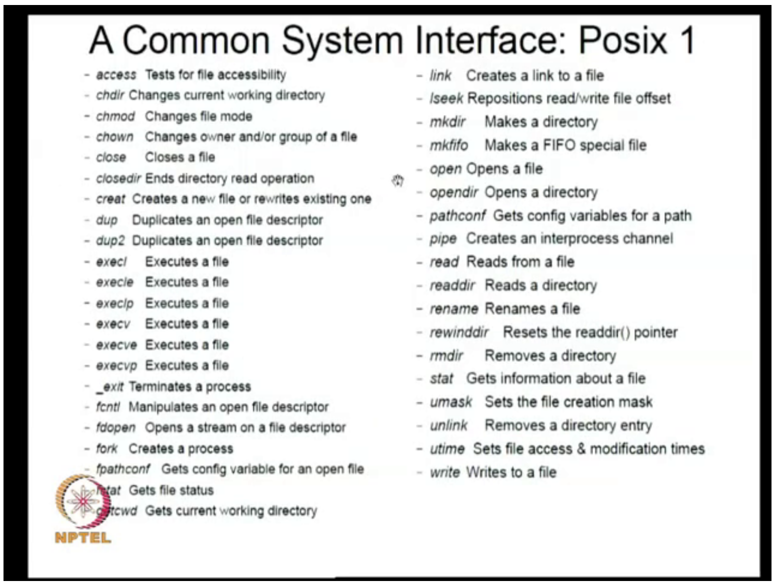
libc

File access
- *fopen* opens a file
- *freopen* opens a different file with an existing stream
- *fflush* synchronizes an output stream with the actual file
- *fclose* closes a file
- *setvbuf* sets the buffer and its size for a file stream
  - *setbuf* sets the buffer for a file stream
- *fwide* switches a file stream between wide character I/O and narrow

Direct input/output
- *fread* reads from a file
- *fwrite* writes to a file

I think we are looking at interfaces the previous class.

(Refer Slide Time: 00:27)

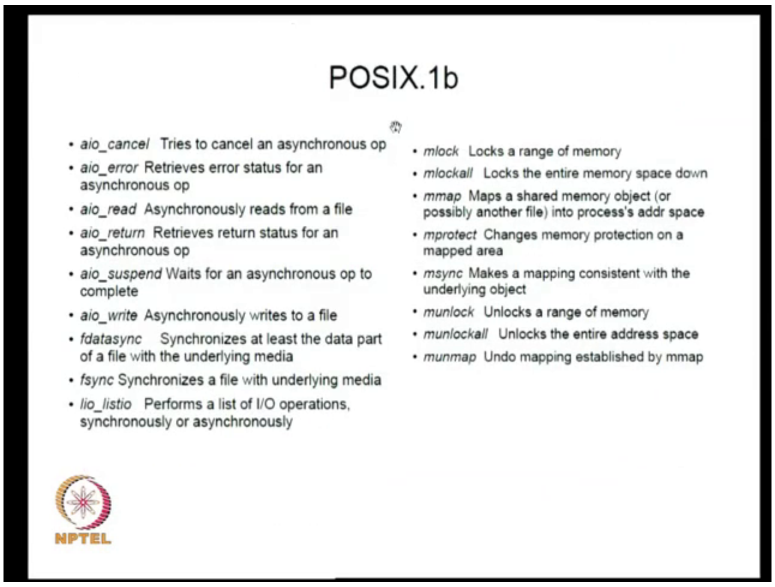


A Common System Interface: Posix 1

- *access* Tests for file accessibility
- *chdir* Changes current working directory
- *chmod* Changes file mode
- *chown* Changes owner and/or group of a file
- *close* Closes a file
- *closedir* Ends directory read operation
- *creat* Creates a new file or rewrites existing one
- *dup* Duplicates an open file descriptor
- *dup2* Duplicates an open file descriptor
- *execl* Executes a file
- *execle* Executes a file
- *execlp* Executes a file
- *execv* Executes a file
- *execve* Executes a file
- *execvp* Executes a file
- *_exit* Terminates a process
- *fcntl* Manipulates an open file descriptor
- *fdopen* Opens a stream on a file descriptor
- *fork* Creates a process
- *fpathconf* Gets config variable for an open file
- *fstat* Gets file status
- *getcwd* Gets current working directory
- *link* Creates a link to a file
- *lseek* Repositions read/write file offset
- *mkdir* Makes a directory
- *mkfifo* Makes a FIFO special file
- *open* Opens a file
- *opendir* Opens a directory
- *pathconf* Gets config variables for a path
- *pipe* Creates an interprocess channel
- *read* Reads from a file
- *readdir* Reads a directory
- *rename* Renames a file
- *rewinddir* Resets the readdir() pointer
- *rmdir* Removes a directory
- *stat* Gets information about a file
- *umask* Sets the file creation mask
- *unlink* Removes a directory entry
- *utime* Sets file access & modification times
- *write* Writes to a file

And I was trying to look at the POSIX interface, I think as I mentioned there are multiple aspects to Posix.

(Refer Slide Time: 00:41)



POSIX.1b

- *aio_cancel* Tries to cancel an asynchronous op
- *aio_error* Retrieves error status for an asynchronous op
- *aio_read* Asynchronously reads from a file
- *aio_return* Retrieves return status for an asynchronous op
- *aio_suspend* Waits for an asynchronous op to complete
- *aio_write* Asynchronously writes to a file
- *fdatasync* Synchronizes at least the data part of a file with the underlying media
- *fsync* Synchronizes a file with underlying media
- *lio_listio* Performs a list of I/O operations, synchronously or asynchronously
- *mlock* Locks a range of memory
- *mlockall* Locks the entire memory space down
- *mmap* Maps a shared memory object (or possibly another file) into process's addr space
- *mprotect* Changes memory protection on a mapped area
- *msync* Makes a mapping consistent with the underlying object
- *munlock* Unlocks a range of memory
- *munlockall* Unlocks the entire address space
- *munmap* Undo mapping established by mmap

This is the POSIX 1 that is also a POSIX 1b. That is also a POSIX 1 c, etcetera, and the various additions. This is the one which is very let us say came early. POSIX 1 b for example, has things like asynchronous I O and things like memory related locking.

And then POSIX 1 c came with threading, multithreading. So, I am now discussing only those things that are connected with storage systems. I think some of this I think we looked at it briefly. For example, we talked about opendir, closedir, readdir, rewinddir, right. So, basically you can take a look at a directory, it can be quite large. And you can actually go through one entry at a time. So, you can open the directory, and then the system will give an interface by which you can read one entry at a time. For example, if I say read directory. It will give me one entry and when I can and it will automatically increment a pointer to the next entry.

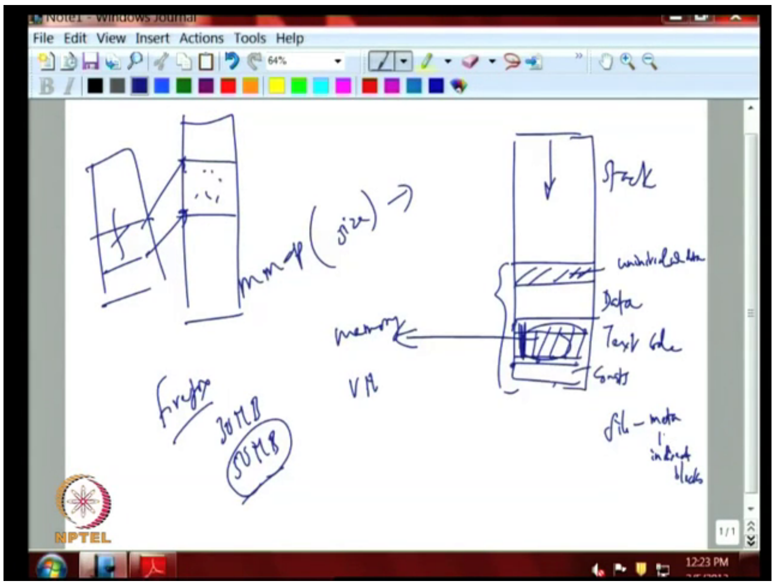
So, that next time when I say redirector I will get a next entry. Now I am not I am insulated from how the directory itself is organized. As I mentioned if it is a very large directory, you can be probably using hashing and other techniques to access these entries in the directory. So, by having this kind of a interface, you do not have to know how the directory is set up. Basically, in UNIX a directory is nothing more than a file itself. So, the only thing is it has got some structure, directory level structure if you got entries, but how will be it is organized is left of the file system.

So, you want to be independent of the file system special file system, you have these things like opendir. So, that you get a initial pointer to the first entry, when you can say given that pointer keep reading it. So, I keep on sequencing over the UNIX over the entries in the directory. Of course, rewind directory will have some peculiar semantics.

Possibly, it might it basically resets the re directory pointer so that you can go exactly to the place where you want. And there are various other things that you may have to I think many of you are familiar with for example, makedir it makes a directly once upon a time make directory was not a atomic operation. There is to be race conditions because of these things. Therefore, it became later an atomic operation.

I think when you talk about exact kind of functions, you notice that it executes a file you might wonder; what is the connection between this and storage systems. So, let us quickly look at that. So, see so if you look at a executable, you find the executable as let us take this as executable.

(Refer Slide Time: 04:10)



It might have some constants, some text, but basically means code. It might have what is called data, and essentially it is also what is called uninitialized data, un initialized data. This uninitialized, data means it can be initialized from values from beginning itself.

Text and some constants we got. Now this is the actual code, I am again simplifying, but this is actually the object code. This is basically the executable, when you load into the system, it will also have a stack here. Stack is going, now what happens when you do an exec is that there are 2 possibilities. You can take the whole of this code, and put it into memory copy it. This is to be done in old systems which do not have what (Refer Time: 05:39) calls virtual memory. Do not a virtual memory you basically read the whole text segment, and put it into memory and execute it.

Suppose you have virtual memory, then what you can do is; you do not have to copies of the strap you just map it to memory. If you have once a virtual memory, what you do is you use some call similar to equal onto mmap. What it saying it says mmap from this point up to this point, map it to some memory location. Starting from this to this let us say the size we map it this whole size into some memory location. So, this this is the file, this is a file, and if this is the address space corresponding to that thing I am just overlaying on top of it. This will be the only the part of the file, this part is the file and (Refer Time: 06:38) here.

So, what you would like to do is; you want to you want to have a memory area, and that is the memory whole area, and this will be mapped on top of it. So, basically what you are going to do is; some of these pages may not be available in the beginning. In the beginning only some may be available, as you access them they will get page faulted in. So, basically what you can do is; you do not have to copy the full text part of it that the code part of it from the file into memory first. You can just start with a few pages, start with a few pages.

You map it, but you map the whole thing. You have to map this whole section, this whole section. You map it and then basically what happens is that, but you are not really reading. Everything you just said that this part of it of the file corresponds to this part of the memory. The memory is not back by is backed by the file, but it is not still resident. That information is still not resident. You start with effusive pages in the beginning and then as you keep going around, you will get page faults and then it will be picked up from the disk on to memory. What is good? What is the idea about this? The idea is that you have a large program, let us say your firefox. Firefox has what is the size of the executable; it can be substantially it could be 30 megabytes, who knows it is nowadays 50 megabytes also.

You notice that if you want to do if you do not have virtual memory; you have to read the whole stuffing. And 50 megabytes reading will take a at least a second or more depending on it. So, there is some delay in loading time, before because the responsiveness of the application is not that much. So, what is the other idea here instead of getting for the full thing to be loaded, I just map it and then as I execute, I will be traversing various portions of the program code. Wherever I will be branching doing various things depending on where I go I load those many things.

So, one thing is I am not loading a full thing, I am loading time is produced other thing is I am using the memory better. Because I am only loading those portions, which I am visiting as execute the code. For example, if you take some very large applications, there were a lot of stuff, and you might not execute all of them. You may execute only small portion of that. For example, it may be that your editor has some support for encrypting your file and decrypting your file. But most people do not use encryption decryption of files.

So, you sitting in the address space in the file has some piece of code that will be executed only if you are interested in encryption and decryption. But most people do not see it. Suppose you have something like 10 other facilities in the program. Most of them are not used only one or 2 are used. So, if you take the previous approach of not mapping, then you will basically essentially stress the memory system. You bring everything in and use only a few of them.

Whereas we use this notion of mapping, and use the virtual memory subsystem to pull in whatever is needed as I execute. I am interested in cryptography it will fault on the cryptographic code. It will bring it from disk again it is backed by the file. So, basically, I have create what is called a mapping from the file to a memory region. So, essentially it is basically, you have this file, it will be mapped to this will be the memory region, and this part of it becomes a text part of it will mapped here somewhere. And you when you pick up those pages as needed, you do not take everything.

So, this turns out that essentially the kernel has to do this mapping for you. That is why there is a specific system called (Refer Time: 11:00) it does out on a other things, so that the process can start. So, the exec v exec various exec kinds of calls are basically about trying to automate this process. So, it requires some help from the file system. That is reason why it is part of I am including it in this part. So, I think most of these sayings you have to look at it by yourself. There is lot of details. I think I even a fork for example, it creates a process. When you create a process, it turns out that you have to open and close certain file descriptors, or you will have to do something it is relating to what is called copy on write.

So, all these things typically take support from the storage system. So, that is reason by this also is part of this set of functions which are related to storage systems. So, other reason why the fork is connected with the storage system; is that you often have the products called swaps face swap; when you create a process you may want you may you will have create the duplicating the swap possibly. Or you might do what is called copy on write on that one. So, to take care of all these things again a file system has to come to the picture. Now let us briefly look at the other part of it, which is basically POSIX would not be; where you are going to provide some additional capabilities. And 2 sets of capabilities you can see clearly see; one is on what is called asynchronous I O other one is what is called relating to locking of memory increasing memory readings.

So, again so what is this asynchronous I O? Basically, there are 2 ways of doing things, if you want to get parallelism. Either you have what is called multiple threads on each thread it has asynchronous reads, but because you have so many threads, there is parallelism. They are all doing synchronous reads; that is, I request something I wait for you to complete. But since I have multiple threads, each one of them can read and it can get blocked they does not matter, but other guys are still running.

So, either have a threading system, or I have a asynchronous system. Now the threading system works out quite well in a user space. Because anyway I need those kind of models. The threading system in the kernel itself may be slightly more non-trivial to make it happen. That is why instead of going for a thread a multi-threading the kernel, sometimes people take the different tack, that is why I will making my kernel multi-threaded is going to be a bit complex.

So, instead of making the kernel complex what I will do is; I will provide asynchronous I O calls. What does it mean it means that, someone in application makes a request for an I O. And then the kernel basically we request a asynchronous I O, and a kernel basically what does it do it basically creates a way in which it can initiate it asynchronously, and then when it finishes there is a callback by which it gets control essentially it gets interrupted. And then it is able to know report back what happened to the user program.

So, you can support it by using certain a certain simpler facilities in the kernel. So, once you make those facilities available, then we can make the asynchronous I O available to user level. For example, if I say asynchronous I O read what does it mean? It means that I initiate the process, but it is going to take a at least some amount of time before like my if I am using a disk for example, it will take some time (Refer Time: 15:29) can get to that place it might be about 25 milliseconds we will say.

So, while that is going on, I can do some other things, and then later I can say, give me the status of that asynchronous operation. I can use this call. Now this is at the user level is aio read an aio return, but correspondingly in a kernel when I do aio read, you will initiate the read, and then provide some kind of a agent which watches over when that read completes. So, when the read completes it will interrupt the system, and then you will couple it with the callbacks routine which will be look at what has happened. And then finally, it will keep the status in some place. When you call aio return again the kernel looks up that status that was there some time back after that operation right. And then it will looks it upon receive that information.

Similarly, aio write you can asynchronously write to a file. You can wait for example, sometimes you initiate asynchronous operation, and then you do other things and then you come back and check if that thing completed. But you discover that it is not complete. Then you have other two choices, either you have other work to do or you really do not have any other work to do. If you do not have any other work to do a best thing is to suspend yourself, that is what this is this aio suspend is basically about. In case you initiated asynchronous operation, you did whatever else you could do, while this operation was while you are waiting for this to happen. But you are through with everything nothing has remains to be done someone has to suspend you also. That is what this is.

Similarly, you have aio error, and retrieves an error status for a asynchronous operation. You can also try to cancel it. It is not guaranteed that you will cancel it. Sometimes it can be done sometimes it cannot be done. For example, if you ask for a read of a for a sector right, the minute to give it will start doing it. And then just a few milliseconds or nano microseconds we have to say cancel it is not possible, because the disk does not recognize that. But if it is a long thing for example, you writing you are doing something like a 4 megabyte or 10 megabyte or one gigabyte it is not right. Right, you can in principle it is possible for you to cancel.

Cancel the part, which is still driven together, because the long enough operation that is possible. So, that is the part about asynchronous I O. Now this is there in POSIX 1 b, it was not there in the earlier versions of Posix. There are also other aspects relating to synchronization. If you do fsync I think we discussed that you have what is called metadata and data in fsync, basically what happens is that you have to synchronize both the metadata; as well as data you have to flush everything to disk then only you can come out.

If you say fsync, every single thing that is sitting in memory has to be synchronized. So, when you want to be clear that you can shutdown the system, you should try to equivalent of fsync or somebody has to do it for you fsync. Whereas, you do fdatasync you are only concern about the data part. Again, this database kind of people for them the data is important, the money amount you have kept in a bank whatever. That is important the if it turns out that account to a full sync whatever reasons, they can at least try to do a fdatasync and at least get the most important things.

There is also some other operations like doing a list of I O operations; that is, I am asking you to read this this this this I can make a list, I can do it issue it synchronously that I wait for a whole things to all the list to be completed or I can initiate each of them asynchronously. That also is possible; then other types of things relating to mappings and lockings corresponding to certain regions. Now you have mlock for example, locks a range of memory. Basically, it is needed in case, if I got some piece of information I do not want you to be moved out till I am completely using this used of that information. So, basically it is required because, you can if you do not do mlock sometimes you get into deadlock situations.

So, basically let us say that, we will take an example, suppose I have the following situation. I have a file; file has got metadata, right? And the metadata basically is in terms of indirect blocks. Indirect blocks, what does it mean? It means that I have a file if you want to access the blocks of it, I go through an indirection scheme; so that I can access the blocks. Not about the read the things, I have to have this indirect block in memory, but indirect block memory is basically also in paged memory.

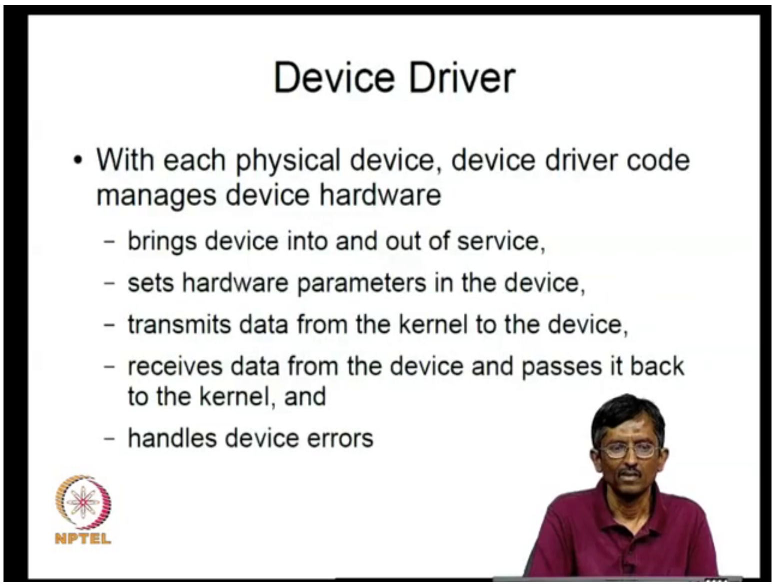
Somebody can essentially push it out. Let us say it is possible. Then what will happen? I am trying to read something, I need the indirect block in memory before I can do any operation to actually go and get the blocks correspond that. But I bring it in before I am done with it somebody is coming and pushing it out, let us say it is possible. There I have a situation where I am not making forward progress. So, in those kinds of situations, it may be worthwhile for me to lock it in.

Now, I am talking about in terms of kernel memory right now. But you can come up with examples and user level also. If this happened user level you better lock it now so that nobody is able to take it away, from you if somebody takes it away from you then again you will have to bring it in and who knows there is a condition in the system that you brought it in, but somebody again taking it out from you. This keeps on happening you get into a problem. So, you and then you have high level idea about what issue is, but typically sometimes you need to lock it down you lock it down you get a you will be able to guarantee that some used from a critical way unit is actually is (Refer Time: 22:45).

So, all right so basically you have you also have something msync, we basically if you have a mapping and you update the mapping. It may not be flush to disk is similar to fsync. What you want to do is you want to make sure that the mapping of the file whatever it is. If something has been updated in memory, but it is not their corresponding on the disk right, you might want to see msync on that. So, there is mmap already mentioned. Usually you map a shared memory object or possibly another file into process address space. And that is what is happening in exec. When you do an exec, you are basically mapping the file into the address space. That is what that is all we do now.

So, if you have locked it you can the corresponding munlock is here. If you have an mmap there is a corresponding munmap. So, it turns out from some of this things there is usually some support needed by the support by the storage system depends on the; because depends on the backing if the backing is on a disk or a file then all these things will have some connection with the storage system. If the backing is coming from some other things and connects with the storage system of course, then it is a different storage.

(Refer Slide Time: 24:23)



Device Driver

- With each physical device, device driver code manages device hardware
  - brings device into and out of service,
  - sets hardware parameters in the device,
  - transmits data from the kernel to the device,
  - receives data from the device and passes it back to the kernel, and
  - handles device errors

Now let us quickly look at device drivers. So, what is the device driver? Basically, for every physical device you need a device driver that manages it. For example, you have to bring the device initialize it, or you may want to stop the device, that also both these functions. You may want to set the hardware parameters as a device, or in a sense the device will tell the kernel these are my parameters, the kernel has to note it. Or the hardware the device has to be told certain things. For example, it may be that the kernel uses certain ways of allocating memory, in certain sizes. You may want to tell that device driver these are the sizes. You needs of you need some memory that you can get hold on.

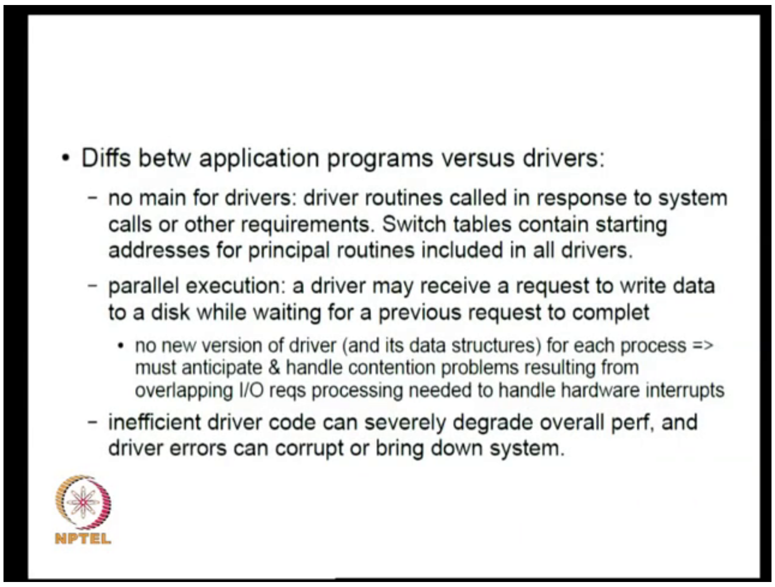
So, basically there is 2-way communication. The device has to inform the kernel about some parameters. Special parameters it has got. Example; in olden days you had something called floppy with 1.44 megabytes, or it could be 2.88 megabytes, it is double density. It could be who knows triple density or whatever. It is a same controller, but there is some once the controller, once you put the floppy in. It knows whether it is single density or double density. The CPU does not know anything about it, only the controller knows it. The controller has to know inform the kernel that here is a double density guy or single density or whatever.

The device driver knows about it, done it actually says if double density it might decide to probably access it in, is it might use certain more compressed models which will be used to read the read the things. Again of course, this obvious thing it has to do also is transmit data from kernel to device. If you want to write, retrieves receives data from the device for reading and gives it to the kernel so that finally it can go to user address space. And were any errors somebody has to handle it. It can be momentary errors or it can be serious errors. Momentary errors means for example, on the bus sometimes there is a conflict. You try to access something somebody else also he is also doing something which prevents the first access to go through you may have to back off.

So, you might get an error saying that it is no longer free. Or whatever because it could be shared to a source, and you have to adjudicate who actually gets use it. Somebody could be partial using it, and there is some period of time it is not being used, and then you have to it may be that you try to use it and you get a busy signal saying that it is not currently available.

So, these are invoke case errors there could be almost serious errors, bus errors with involving like there are electrical errors. So, basically that termination, there is some electrical signals traveling back and forth, and those depending on the way the transmission line or the resistance and capacitance on that line there could be some problems. So, you try to see if there is a way to recover from it whatever you have, if it is not possible, you have to just say (Refer Time: 27:57) I can not take the I can not do anything with this all the stuff is there.

(Refer Slide Time: 28:00)



- Diffs betw application programs versus drivers:
  - no main for drivers: driver routines called in response to system calls or other requirements. Switch tables contain starting addresses for principal routines included in all drivers.
  - parallel execution: a driver may receive a request to write data to a disk while waiting for a previous request to complet
    - no new version of driver (and its data structures) for each process => must anticipate & handle contention problems resulting from overlapping I/O reqs processing needed to handle hardware interrupts
  - inefficient driver code can severely degrade overall perf, and driver errors can corrupt or bring down system.

So, basically if you look at device drivers, there is no main routine. Basically, it is a bunch of code which is input inside the kernel. And it is basically giving you some services, how to read the device how to write the device, how to manage the device. So, and basically all the routines that the driver provides is a ways; so that the kernel can call them. For example, there could be on a tape, something called a rewind kind of a command. So, then the kernel can say rewind, and then the controller will essentially parse it, and know that you talking about a rewind, and then it will basically command the device to go to the beginning if it is a tape.

So, there will be some set of functionality that is available for each device. Normally this functionality is kept as far as possible generic. For example, almost all devices you will have something called open the device, they will have something called close the device. They will have something called read the device, something called write the device etcetera these are all, but also a few other things also, but that is particular to some the specificity of the devices. There will be some additional commands, but there is a core set of things are typically always available.

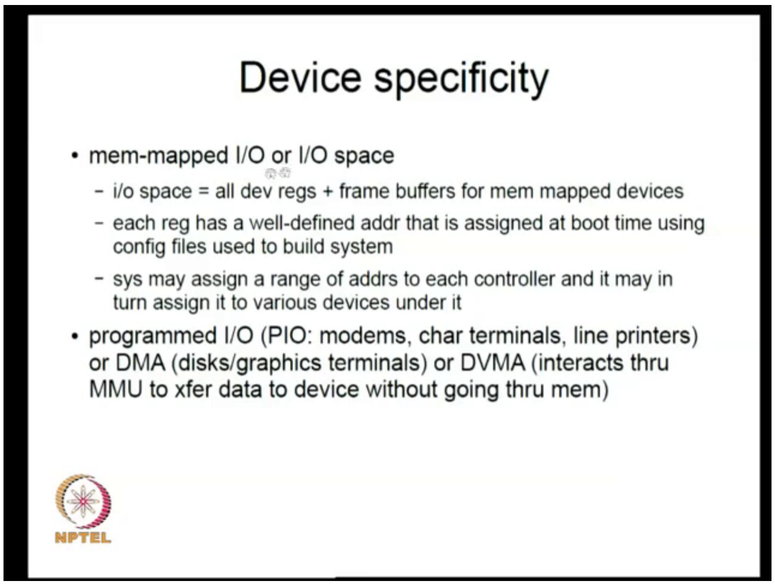
Other thing about drivers is that typically parallel execution is expected. Except for very simple device drivers, extremely simple device drivers it might do everything serially, but most device drivers parallel execution is something which is taken for assume to be typically the case. Basically, what is a reason? Why because devices are slow, if devices are slow compared to CPU, you have to keep the devices as full busy as possible. Any parallelism that will help it will keep all the device busy is beneficial. From the from the performance point of view; that is why a driver may receive a request to write data for a device while waiting for a previous request to complete.

Now, what you have to do? Do you create another instance of the driver? Or you handle, basically have basically you can handle this contention problems. So, basically because there could be multiple interrupts coming in. You have to basically handle it make sure that this particular interrupt is for this particular request; you need to match all those things. So, and this is not really under the controller program, because interrupts are coming from outside. There is no control and the only control we have is to disable interrupts that is the best you can do.

But that is to grow some measure. So, the thing is it may be that there are multiple devices who are actively accessing multiple disks. And each of them will be completing at different times, and they will be interrupting the CPU, and you have to figure out whose interrupt is for whom, and then actually do the right things. So, there is lot of parallel is parallelism you know device driver it is possible. And basically, inefficient driver code can basically destroy the system. I think a lot of you have seen sometimes that there is something stuck in the floppy or some other thing or a CD ROM called the guy is just sitting and not allowing you to proceeding system also.

You must have seen that happen. That was because it is happening at in the in the kernel, but there is some problem and a kernels trying to fix it or trying to figure what to do during that time it is not accessible to anybody else.

(Refer Slide Time: 31:43)



Device specificity

- mem-mapped I/O or I/O space
  - i/o space = all dev regs + frame buffers for mem mapped devices
  - each reg has a well-defined addr that is assigned at boot time using config files used to build system
  - sys may assign a range of addrs to each controller and it may in turn assign it to various devices under it
- programmed I/O (PIO: modems, char terminals, line printers) or DMA (disks/graphics terminals) or DVMA (interacts thru MMU to xfer data to device without going thru mem)

There are lot of differences in devices, you can have what is called memory mapped I O or you can access something called I O space; that is, a instead of memory mapped, you have an I O space, right? And you may if it is memory map what it means is that, there is in the address space there are specific locations which map the registers of the device. So, the CPU just has to use that memory, memory values then everything is fine. If it is the I O space you have to use some specific instructions. For example, in Intel there is something called I think it is called I O byte I O w something is there basically you can there are ports, and you tell it which port it is and then you basically say read and write from that.

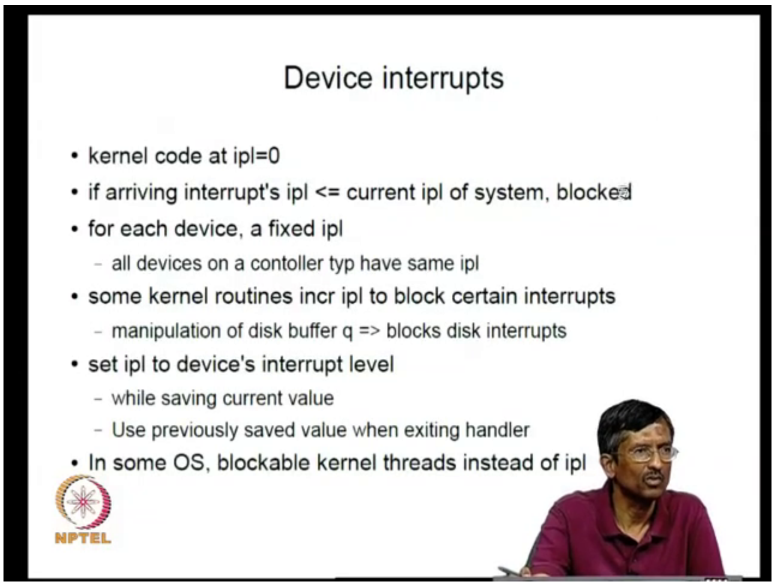
So, they do not use the memory instructions. They have a specific set of instructions, which only deal with I O ports, and then you have to say which port it is, and then you have to say which a part of that set of registers that are there in the device access through that port. You have to say which one of them it is. So, there are 2 types a memory mapped and I O space. And these things for example, if it is memory mapped when the system when it is booting will assign a range of addresses to each controller. And this controller might be for example, I might have one control and managing multiple disks.

So, the controller gets small bunch of addresses, and that in turn can be given to each particular devices, that also is possible. So, there were a other models like programmed I O, I think basically that the CPU takes direct control of the sending of the instruction and getting the basically it is waiting for it to complete. There is a using DMA; you basically have an autonomous engine that is basically doing it for the CPU. And in this case DMA it turns out it is using typically physical memory, whereas you can also use what is called DVMA which uses virtual memory.

So, there are some complications if I you go either this route or this route. The good thing about going with a DVMA is that a CPU's does not have to worry about the physical mappings etcetera, because it all deals everything in virtual memory. But then the devices has to worry about how to access virtual memory. Because it has to equivalent it in a essence it needs a mapping also. The devices are known it is a mapping so that you can talk about what is there if you do this physical memory, the DMA programs using physical addresses, then there is a problem with it because typically these devices have only some amount of address space, that 24th bit or whatever it is. And that might be quite different from what is there in a CPU.

For example, I might have right now a 64-bit CPU, but the device because it is designed for a particular purpose. It may require only a smaller number of address bits. So, it might not have 64 bit addresses it might have only have much smaller one probably it might have only 28 bits or 24 bits or whatever that means that somebody has to figure out how to make sure that whatever it is accessing you now fits into the virtual outer space of the CPU. So, there is has to be some adjustments made and there as a lot of device specificities that are involved here you have to figure it all when you writing the device.

(Refer Slide Time: 35:38)



Device interrupts

- kernel code at ipl=0
- if arriving interrupt's ipl <= current ipl of system, blocked
- for each device, a fixed ipl
  - all devices on a contoller typ have same ipl
- some kernel routines incr ipl to block certain interrupts
  - manipulation of disk buffer q => blocks disk interrupts
- set ipl to device's interrupt level
  - while saving current value
  - Use previously saved value when exiting handler
- In some OS, blockable kernel threads instead of ipl

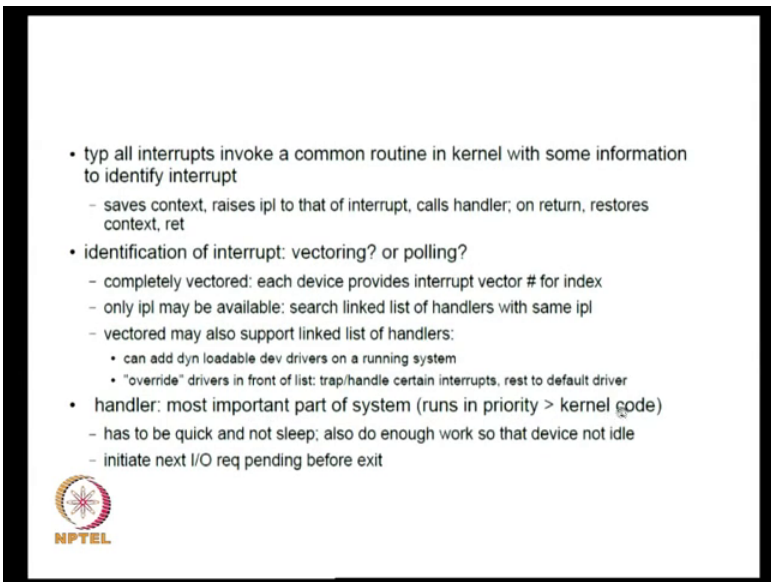
There is also interrupts, right? So, let us say that interrupt party level of the kernel is 0. Then basically you might if the arriving interrupt ipl is less than the current ipl of system you block it; that means, that you basically have a particular level, and if an interrupt comes only above it you let the interrupt go through. Otherwise it is blocked. It is kept in abeyance and later when you if you reduce your interruptible then it is actually look at later. So, for example, some kernel routines, you have to increment the ipl to block certain interrupts manipulation of disk buffer q.

Basically because if you look at it some interrupts, they manipulate the disk buffer q now the kernel also manipulates a disk buffer q. Now a both the things used for the kernel while it is manipulating this buffer q when interrupts comes in then it is possible because it is it can actually they can be a loss of synchronization mutual exclusion sorry, there is if there is there are mutually exclusive. The mutual exclusion is not, there then it can corrupt the (Refer Time: 36:8) structures; so the serial why you will find that certain kernel routines. If they are manipulating this buffer qs. They might say that I do not want any interrupts from which will actually also have something to the disk buffer queue probably that disks only.

It might decide that the disk in interrupts relating to disk have to be blocked. They might do those kinds of things. So, typically for each device there is a fixed level, and all the devices on the controller. For example, a controller might have multiple tapes for example. One single big controller might be dealing might be managing multiple tapes tape controller. So, typically all the devices on the controller have the same ipl, same now. These are usual usually the case. So, usually when you set the ipl to device interrupt level; while saving a current value, and then when you exit the handler you basically save that previously saved value.

So, you basically restore it backs only, you restore it back. And it turns out in some cases, there are in some operating systems, instead of doing this blocking, they have what are called blockable kernel threads. For example, solar is does this this is slight a different design compared to this one.

(Refer Slide Time: 38:18)


- typ all interrupts invoke a common routine in kernel with some information to identify interrupt
  - saves context, raises ipl to that of interrupt, calls handler; on return, restores context, ret
- identification of interrupt: vectoring? or polling?
  - completely vectored: each device provides interrupt vector # for index
  - only ipl may be available: search linked list of handlers with same ipl
  - vectored may also support linked list of handlers:
    - can add dyn loadable dev drivers on a running system
    - "override" drivers in front of list: trap/handle certain interrupts, rest to default driver
- handler: most important part of system (runs in priority > kernel code)
  - has to be quick and not sleep; also do enough work so that device not idle
  - initiate next I/O req pending before exit

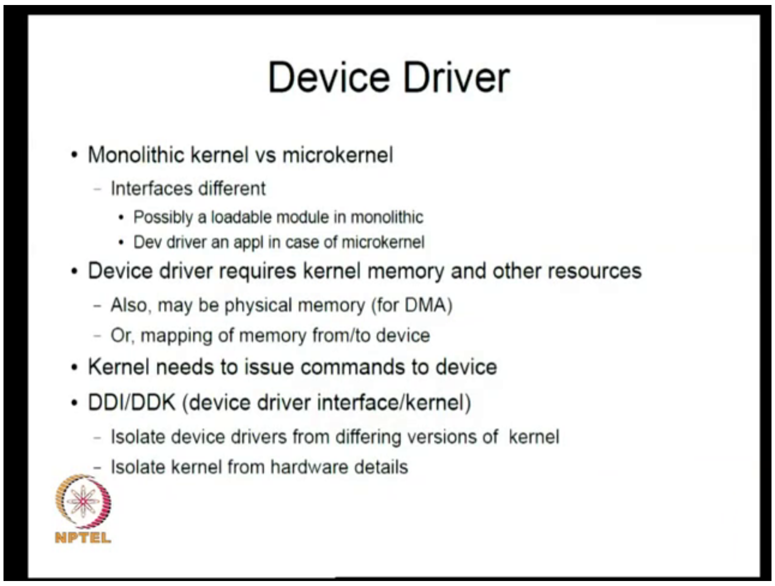
We will not get into try it now, but there are different designs possibility are also. So, even there are quite a few possibilities here. Typically, all interrupts invoke a common routine in kernel with some information to identify the interrupt. You have to identify the context, you raise the ipl to that of interrupt calls the handler on return restores the context and returns it.

So, various things happen. But even here there are lots of differences, identification interrupt you was doing when you call vectoring or a polling. So, there are multiple possibilities here also. So, usually the handler, is it quite an important part of the system, because it runs in a priority bigger than that are the kernel. Because they are running a higher priority than kernel, it is you have to make sure it runs only for a short periods of time. In something the kernel, the supervisor, if there is somebody who is a bigger than supervisor is in which the super weather is not able to work he is managing a whole system right.

So, you have to ensure that this handles have to be quick, and it should not sleep. If they block then the whole system can get blocked. So, these handlers they have to ensure that, if they require in a memory they pre-allocating beginning they do not. Try to allocate being a interrupt handler. The interrupt handler knows that it requires memory, you better pre-allocate it you get it beforehand keep it. With you do not try to do it at runtime. You do it much at here do not do it at the time when interrupt happens.

So, this is something which is a quite an important part, you may right interrupt handlers you have to ensure that they do not sleep very critical. Again, other thing is that you finished one part of the processing, you will check whether one more pending request is there. If one more pending request is there, you initiate that thing and go. Basically, in some sense you want to keep the device completely busy, something finished, you try to handle it, you handled it, and then you notice that after finishing it there is still one more request for the same device. Then you basically again initiate the next I O before we exit all these things have to be done by this handler also.

(Refer Slide Time: 40:36)



Device Driver

- Monolithic kernel vs microkernel
  - Interfaces different
    - Possibly a loadable module in monolithic
    - Dev driver an appl in case of microkernel
- Device driver requires kernel memory and other resources
  - Also, may be physical memory (for DMA)
  - Or, mapping of memory from/to device
- Kernel needs to issue commands to device
- DDI/DDK (device driver interface/kernel)
  - Isolate device drivers from differing versions of kernel
  - Isolate kernel from hardware details

So now there are also differences in device drivers with respect to whether it is a monolithic kernel or a microkernel, intonates are quite different. It turns out the device driver in a microkernel is like an application, almost exact like an application. Whereas, in the case of monolithic kernel, they are most likely be it is called a loadable module. What is a loadable module? It means that I can load it at runtime, and unload it at runtime. I may not have all the support for it in the beginning, but I see the device coming online, and then I will load the corresponding module kernel module corresponding to that particular device, that is called loading and then once I am done with it I can unload it also.

That way I can reduce my memory footprint I reduce somewhat of; so, device driver requires kernel memory in other resources. Also, may be physical memory for a DMA. Because I mentioned, if you have a DMA device, it can only be access it may possible that it only understands physical memory not virtual memory; that means, that if these device are talking to each other, the device and the kernel they have to decide on what physical memory they are going to assume both of them know what. So, that when the device driver writes in the physical memory, the kernel knows where to go and find it.

So, they have to have some understanding. Similarly, you can have device which have memory large amounts of memory. That can be mapped into the address space of the system. That also is possible a good example is you have this graphics cards, the graphics cards have might come with large amount of memory. And the CPU might want to access it. And one easy way to do is take the graphics memory and map it to the kernel address space. And therefore, now the kernel has a way to talk about it both are seems to talking about the same thing. So, when the kernel writes to that kernel address space it is equivalent to being seen by the graphics card. That also is there.

So, there are various interfaces in the system, often times called DDI DDK. One is device driver interface, and there is a device driver interface to the kernel. So, what is this device driver interface? Basically, this is a thing which tells you how to get the resources from the kernel. And this one basically tells how the kernel gives, how the kernel can interact the driver; both ways there are essentially a device driver is you load the insert the kernel and it leaves in the kernel once it is loaded. And since it is inside the kernel, the kernel will provide certain primitives for example, memory allocation.

So, the device driver will need to use those primitives. So, all those things are encapsulated in these interfaces DDI DDK. And they serve for one more reason. One is that you want to isolate device drivers from differing versions of the kernel the kernel can keep changing, I think if you are familiar with Linux we will find the kernel keeps changing all the time you write a device driver about 2 years back. And then if the kernel interfaces changes that mean, you have to go on change your driver also, this is a big problem. So, if you have a DDI DDK that problem is obviated.

In a sense you your driver is return to a particular interface standard, and it is expected with the kernel also will honor that standard. And that is what happens in commercial systems. Linux basically does not have that model. Then it basically says that because the source is available. You can always look at the kernel facilities and re come you basically have to in that your driver. So, that it can use the new facilities. So, from that point of view it is a slightly more challenging to live in a Linux kernel environment for writing device drivers, because it keeps changing that is why the device drivers wrote 10 years from now most like it will not work right now.

Whereas in the case of commercial systems it is usually there are various standards they will say DDI DDK version number 9 or 10 or whatever it is, and you say I am programming it to DDI DDK 9 whatever it is, and they provide usually a compatibility layers so that even if there is a slightly older device if there is a compatibility layer it will somehow do it some translation. And so, that even the older devices come your older device will have a still works with the new operating system. That is possible.

Other reason why you need this is; you want to isolate the kernel from hardware details. Basically, you notice that if you are in operating system, it is wants to read and get some information in and out. It can be through a network, it can be through disks, it can be through what say terminals sorry it not terminals let us say keyboards etcetera, right? From the kernel point of view, it is just a read and write, but each of these things are different. It discuss different a terminal keyboard is different; a tape is different.

So, one possibility is for the kernel to incorporate models of all these things in the kernel itself. That blows of the size of the kernel. I think if you look at Linux for example, you will notice that it is a really big system; it is got some I think over now about 30 million lines. I think the kernel itself is quite small. Most of the space is occupied by file systems and device drivers most of the space. Now it is very difficult to guarantee the correctness of and very large program, written by so many different parties 100s of parties it is very difficult to guarantee whereas, the kernel itself you can guarantee.

So, if you make the kernel know have to know all the details for all the hardware devices, then the kernel slightly be extremely brittle it is not going to be working out very well. So, right way to do it is to say that; I will isolate the kernel from hardware details I have an interface and if I have a very proper interface DDI DDK, then I can check for conformance also. Somebody writes a device driver a check if you is using the interface in the way it is supposed to be done just like c language you know you have a linked program it checks whether you are using the things in a proper way right.

Then if you see that it is being done then you can you know you have some chance that it is going to work some chance. So, if you have the DDI DDK, you can attempt to isolate a kernel from hardware details, and the way it is done by is publishing a set of functions. Like, I have mentioned open close read write, or there is something called ioctl call ioctl call is some kind of a catch all call by which you can give some specific device specific information, and the device driver that is written for it we will look at the device specific information and use it to understand what has to be done.

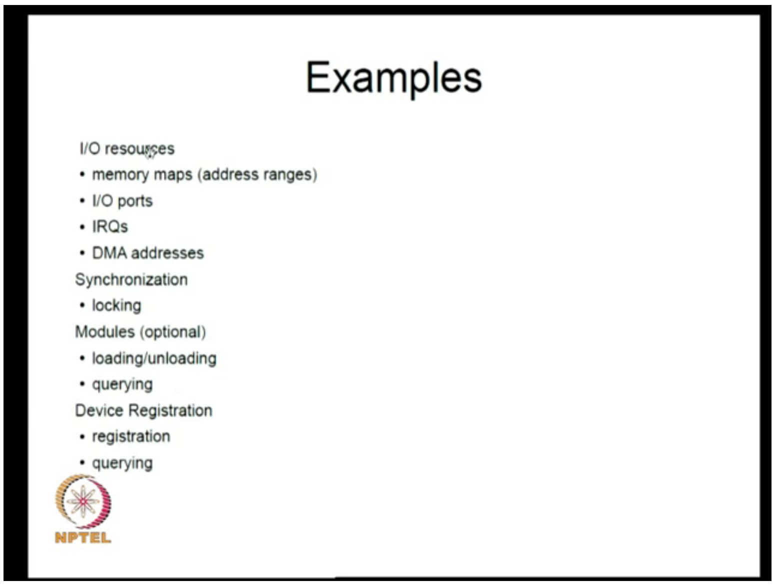
But usually it is encapsulated as read write open close etcetera. So, that all the devices will have the same things, and the kernel only will call these routines; open, close etcetera. It would not call any anything that is specific to the device per say it is only calls this particular interface is available in this DDI DDK. We will look at this. So, we will take a look at one simple device driver so that you get some idea about what is going on.

So, there are if you look at a micro microkernel kind of base systems, it turns out that device drivers will be separated out from the kernel, strict to separated out. They would not leave in the kernel also and that is a very good design in the sense that as I mentioned there are so many device drivers any one of them can be wrong, right? Because it is not written by the kernel hackers written by device driver device people were design the devices, and they may not be as knowledgeable about the kernel, as people who design operating system.

So, if you look at it the devices if we have the wrong device driver or a malfunction device driver they can essentially compromise the system. So, whereas, in a microkernel, what happens is that device drivers are outside the kernel. Because they are outside the kernel, you can essentially is there any bugs, right? The device driver crosses dies, the kernel does not die. So, that is one very strong point for going microkernel, but so far, this experiment as not work very well most current operating system including Linux we are using the monolithic kernel model; where the device divers are actually part of the kernel after loading.

That means that the reliability of the kernel is not dependent on the reliability of device drivers. And so, and because as I when mentioned already, the device drivers may not be written by experts, so there is a problem there. Whereas, monolithic kernels also, but it is somehow because where the other reasons it does not caught on probably in a future you might find some experiments in this direction. So, let us just quickly look at what are the things that the device driver and the kernel they negotiate.

(Refer Slide Time: 50:23)



Examples

I/O resources
- memory maps (address ranges)
- I/O ports
- IRQs
- DMA addresses

Synchronization
- locking

Modules (optional)
- loading/unloading
- querying

Device Registration
- registration
- querying

They can negotiate on some I O resources memory maps. For example, it may be that some devices want to be preferably at particular locations, or it may be that the address space itself has some restriction.

For example, if you look at the old IBM-PC, they have a restriction that all the devices will have their memory locations between one megabyte and some 768 megabytes (Refer Time: 50:56) between that or 640 kilobytes to 1 megabytes. A 640 kilobytes to 1 megabyte. That is again coming from the pc architecture definition not coming from a device part. Whereas, it could be that as I mentioned earlier if your device uses DMA, and it is using only physical addresses, then it may be have only 16 pins; it 16 pins that is usually about how many kilobytes it is only 64 kilobytes. So, it is the device able to access only 64 kilobytes; that means, that whatever it is doing, it has to fit into a 64 kilobyte, area in the address space. It might be starting at 0, or it has to be at a particular offset only 64 kilobyte. That that does the only window it has got.

So now they have to agree on where it is going to be. Or it could be a 24 bit, let us say a device is 25 below addresses; that means, it can access 16 megabytes. Now it has to be now positioned somewhere in the address space of the kernel somewhere. Again, that all these things are to be negotiated. Same thing I O ports how many interrupt request times are there. As I have already mentioned DMA addresses. Synchronization, the kernel will have certain notions of synchronization. For example, if it is a unit processor, it might do with certain types of synchronization mechanisms. If it is a symmetric multiprocessing system, you will have some more complicated etcetera, synchronization primitives.

The device driver now has to use those primitives only. So, because the device driver is now resided in the kernel after loading, it can only use those primitives that the kernel has provided to it. And then so, there is a lot of dependence of the device driver on the locking facilities that are provided with the kernel. And the modules for example, it can be optional it says that some designs will say that you cannot load and unload at runtime. Whereas, many systems will allow you to load unload. So, the Linux allows you to load a unload runtime.

What is the problem about loading and unloading at runtime? You are security maybe compromised. Because somebody can by whatever means has got a access to the system, and is loading some insecure kernel module or whatever. He can create some complications. So, if you are looking for security serious security. If I will disable all this loading unloading parts of it there is also a querying. You want to see whatever loading on the system, because you are it is a dynamic system where load unload things. At some point you want to know what does what does status of the system is that is what you querying.

And device registration basically you have to essentially what happens is that; when you have a device, you might give some you might want to give the multiple devices for example, you might have 10 tape devices attached to it. I want to be able to say which tape device I am talking about. So, there will be a what is called a major class which is about the tape class. And in there will be a minor device number which is basically with specific tape device. It is tape is 1 2 3 4 etcetera.

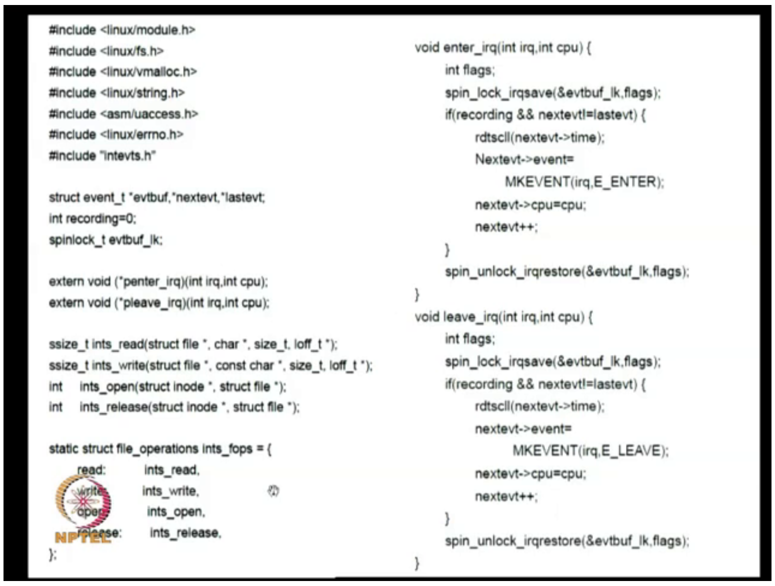
So, you might want to register the device. How does the registration happen? You attach the system is let us say, already you can set up. You power on the machine, and then somebody is going to look at somebody at the initialization time, they will look and see who is on the bus. It will probe it, as it is called. And then it will discover that because a electrical there is some there is going to be some intelligence in the bus system, and is giving a simple example. If it is it will discover there are 4 devices, there then we will say the first one is going to be I am taken a simple system. First of all, we the you will have a minor number 1, second, we will have minor number 2 3 4. Somebody has to do it at initialization time, or it could be done later.

If you are what is called hot swappable devices, I can do it at runtime that is a the system is already running, I can insert a disk or a take away disk. To insert a disk, I need to give it a new minor number a major number is corresponded with disk driver, because it is valid for all disk drivers, but the minor number will be that corresponding to that particular device are inserted right, now and there has to be kept note of basically because I might have a variable number of disk drives. So, I have to keep track of it, when the new device comes in I should be able to say, this is the new number when it is taken out I might use some previous number which is no longer available with that disk is no longer available.

So, I can do all this kind of things, and that is also a facility for querying it you can look at it and say what devices are connected to it. And here there are lot of interesting things that happen, because in a disk you have to be very careful who actually has got a access to it if the wrong person accesses it you can actually cover the disk. So, typically on the disk you often have your own area on the disk which says that; I am so and so, I have grabbed this particular disk device, and putting a specific marker on it or whatever called a magic number on it so that if the magic number is there I know that I wrote it you somebody (Refer Time: 56:34) or something our kind then only is something that happened to it the magic number is no longer there.

So, we need to do some of those things so that at all times you have clear about what is it that you are dealing, with if that clarity is not there you can get into problems. So, a lot of issues out are there at this point and this kind of systems, and what we will do next class is through go to we will look at one of the device drivers.

(Refer Slide Time: 57:01)



```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/vmalloc.h>
#include <linux/string.h>
#include <asm/uaccess.h>
#include <linux/errno.h>
#include "intevts.h"

struct event_t *evtbuf,*nextevt,*lastevt;
int recording=0;
spinlock_t evtbuf_lk;

extern void (*penter_irq)(int irq,int cpu);
extern void (*pleave_irq)(int irq,int cpu);

ssize_t ints_read(struct file *, char *, size_t, loff_t *);
ssize_t ints_write(struct file *, const char *, size_t, loff_t *);
int    ints_open(struct inode *, struct file *);
int    ints_release(struct inode *, struct file *);

static struct file_operations ints_fops = {
      read:       ints_read,
      write:      ints_write,
      open:       ints_open,
      release:    ints_release,
};
```

```
void enter_irq(int irq,int cpu) {
      int flags;
      spin_lock_irqsave(&evtbuf_lk,flags);
      if(recording && nextevt!=lastevt) {
            rdtscll(nextevt->time);
            Nextevt->event=
                  MKEVENT(irq,E_ENTER);
            nextevt->cpu=cpu;
            nextevt++;
      }
      spin_unlock_irqrestore(&evtbuf_lk,flags);
}
void leave_irq(int irq,int cpu) {
      int flags;
      spin_lock_irqsave(&evtbuf_lk,flags);
      if(recording && nextevt!=lastevt) {
            rdtscll(nextevt->time);
            nextevt->event=
                  MKEVENT(irq,E_LEAVE);
            nextevt->cpu=cpu;
            nextevt++;
      }
      spin_unlock_irqrestore(&evtbuf_lk,flags);
}
```

You just go through high level what exactly is going on. Here how is this particular system how does it work. This is some device driver one of my students show it long time back about 10 years back. So, it would not work on current Linux systems, you can probably get into that. But the high-level ideas are interesting simple enough that I can cover it next class. I just go through it, so it will get a fairly good idea what device drivers are similar features are there in current Linux systems you might have to go on study it to see how to run the same thing on this new Linux systems.

But, we will look at it a next class. That is for all today.