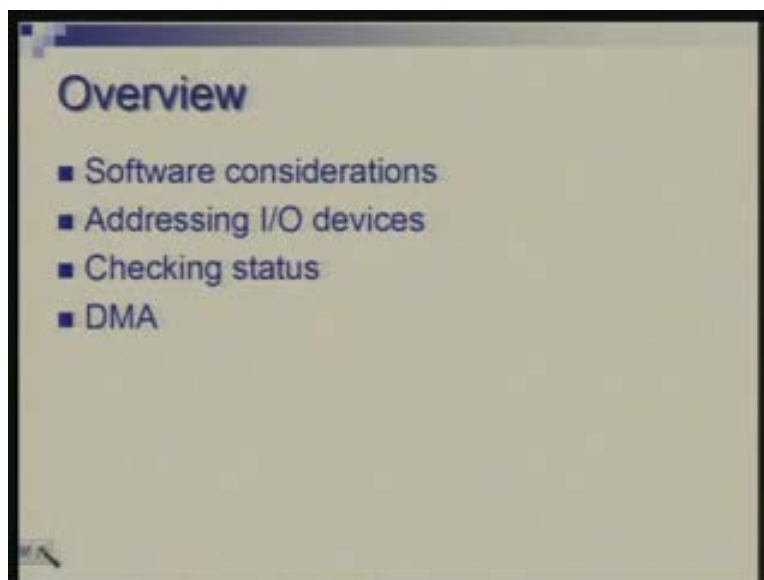


Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 36
Input/Output Subsystem: I/O Operations

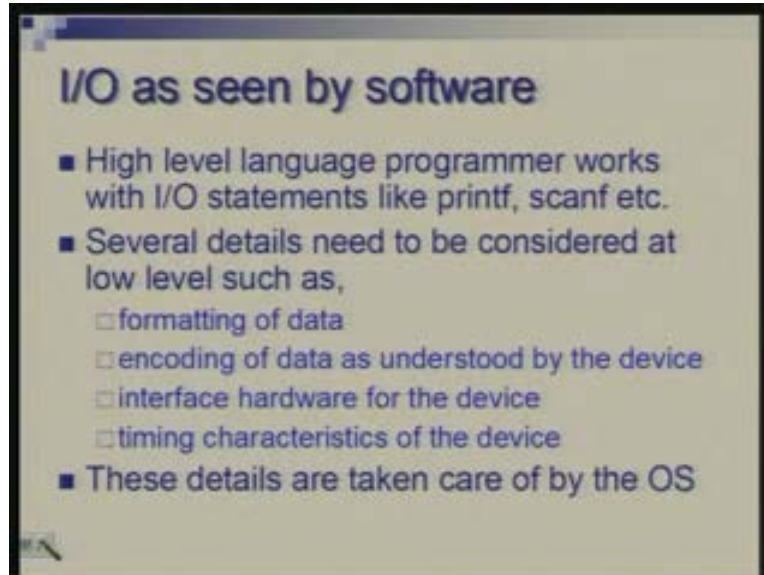
Continuing our discussion on I/O subsystem, we will today talk of I/O operations. So far we have seen peripheral devices and the buses which interconnect them. Now we will try to see how the whole thing works together. So, we will try to view the whole thing from a software point of view as seen by a programmer, how the entire activity appears.

(Refer Slide Time: 1:24)



We will look at the issue of how I/O devices are addressed and one crucial thing in I/O operation is checking the status or identifying when certain events are occurring. And finally we will talk of mechanism which allows memory and I/O devices to talk to each other. Initially we will see from processor point of view that is what I mean by saying talking from software point of view. So we will see how processor participates in this activity and finally how memory directly interacts with the devices.

(Refer Slide Time: 2:04)



So, as seen by the software a high level language programmer tries to see the I/O activity in terms of some abstract statements. For example, we have statements like printf and scanf in C and your view of I/O as a high level language program would be in terms of what these statements can do. So you can send the data out, you can get the data in through these statements; and these are fairly high levels of abstraction.

So once you look at these at the low level at the machine language level there are lots of details which need to be taken care of. For example, you need to worry about the formatting of data. When you are using printf statement **you may** you look at the data as high level language program sees it in terms of arrays and structures and these structures could consist of various types of primitive elements: integers, floating point, strings and so on so how all these has to be converted into raw form of data that formatting has to be done.

The values have to be encoded in the form in which devices can understand and each device will have its own way of representing information. For example, display device understands things in terms of matrix or pixels. A character level device such as a keyboard understands things in terms of ASCII characters and so on. So the information when it is coming in has to be brought in from the form which peripheral device is sending into a form which programmer would like to see and vice versa. Then the I/O operations at the low level have to deal with the interconnection structure; between the device and the processor or memory what kind of hardware is there, what kind of bus is there, so how to get the data through these paths or these structure has to be seen. And then timing characteristics of the devices also has to be taken into account.

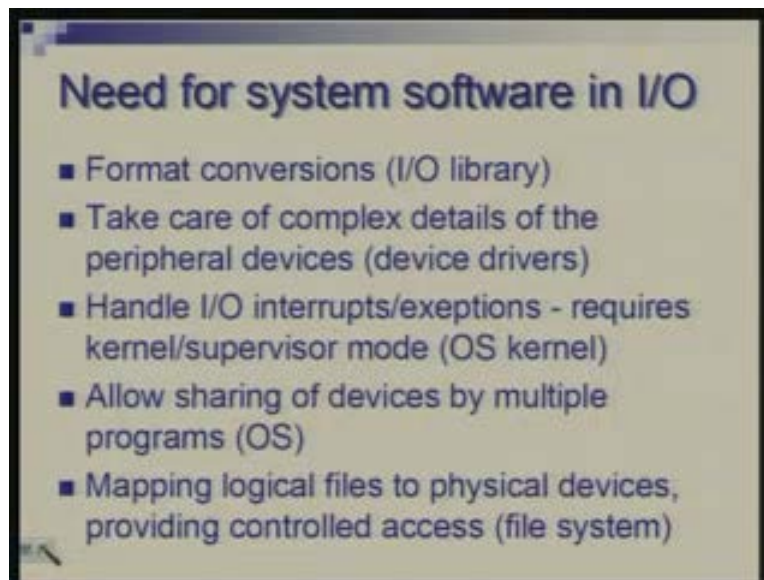
For example, if you are trying to read from a particular disk track, it takes time to have the head move **from that** from the current position to that position so there are certain

timings involved which have to be taken care of. So all these details have to be taken care by the system software and a programmer gets an abstract view.

So, to be more precise what is the role of system software in this whole operation?

You have format conversion for example. You have I/O library, when you compile a program **there is a lot of** there are lots of library functions, library routines which get linked up into your object code so they take care of converting some of the abstract operations into low level operation. So things like format conversion would be handled at that level.

(Refer Slide Time: 04:55)

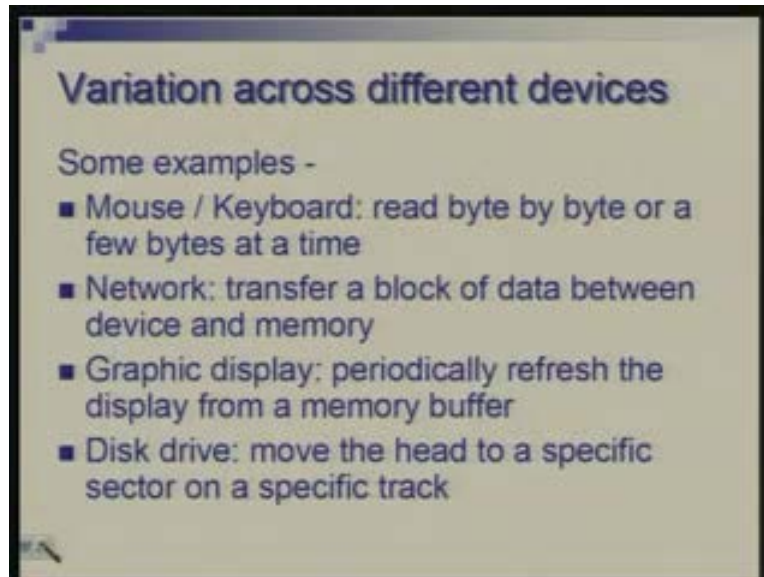


Then there are complex details of the device itself. The issue of dots or the pixels in the pixel matrix or the question of tracks and sectors of a disk drive these details are known to what is called device drive. So device drivers look at specific peripheral devices or a family of peripheral devices and understand the details of how these work, how the information is to be handled and so on. Then you require to work with interrupts. So the events which occur in the I/O devices or their timings they have to be handled through interrupts or exceptions and as we have seen earlier that these are handled by exception routine or interrupt handling routine which are part of OS kernel. So there is an interrupt system, there are number of interrupts which are possible for a given processor and they are handled by a portion of OS kernel.

Then there is also a question of multiple programs or multiple processes sharing different peripheral devices. So, several programs or processors are trying to share a disk or share a printer or on display you have multiple windows coming up due to multiple tasks. So the sharing also has to be looked after and that is **the role** one of the roles which was placed. Then in devices like: base or hard disk drives, floppy disk drives, CD ROMs and so on, the information is organized in terms of files. So you have certain logical view of files as you see in a high level language program. But these files have to be mapped to

physical devices in a certain manner. So you see some records in the file, at the physical level there may be tracks and sectors. So, how this mapping is done is taken care of by the file system part of operating system. So there are all these different aspects which take lot of drudgery of I/O operations away from a high level language programmer.

(Refer Slide Time: 07:23)



Now, the requirement of different devices could vary quite vastly depending upon nature of the device, the kind of speed it has **there could be**..... the demand which these devices place on the system may be quite different.

If you consider, for example, slow devices like keyboard or mouse, then these are producing information which is governed by the rate at which a human operator can enter. For example, keyboard is limited by the speed at which you would type. So the demand is that you are transferring at times only one character, 1 byte at a time or maybe a small group of bytes at a time and the rate at which this happens could be just a few bytes per second.

Same thing you could say about mouse. The mouse basically is tracking the position and as you move the mouse from one place to another place essentially what is being sensed is the movement; whether it is in plus x direction minus x direction plus y or minus y or a combination of them. So, from one position to other position the movement is slow. Imagine that the processor is working at the rate of several gigahertz and the mouse movement is in terms of probably a fraction of second. So the rate at which the information is being generated is pretty small. You are handling small volume at a time maybe just a character or a few characters and at a small rate.

On the other hand, if you look at network as a peripheral the rate could be fairly fast and you also look at a block of information. You are not talking of sending a byte on the network or sending 10 bytes on the network, it is always in terms of big blocks or

packets. So the way in abstract terms you would like to see network operation is that you have a block of data in memory, you want it to be sent out on the network or you have a block of data coming from network code, you want it to be placed in a particular position of memory.

So the rate could be for example, the network standards talk of 10 megabits per second or 100 megabits per second or 1 gigabits per second, now we are going to 10 gigabits per second which would mean typically a few hundreds or few tens or hundreds of megabytes per second.

Now let us turn our attention to graphic display. In graphic display what you want is some picture or some text or some combination of these which is steadily displayed on the screen. That means the information on the screen has to be refreshed so that a human eye sees it as a constant image. So you need to refresh it several times a second, could be twenty five times a second or thirty times a second and what is done is that you maintain an image of what you want on the screen in memory. It would be called video memory and this memory would be within the address space of the processor.

So basically any change you want to do on the screen, you modify certain area in the memory which is directly reflected on the screen. So there has to be some activity which takes matrix of pixels as stored in the memory and refresh certain number of times per second on the screen. So this is how most of the modern systems work.

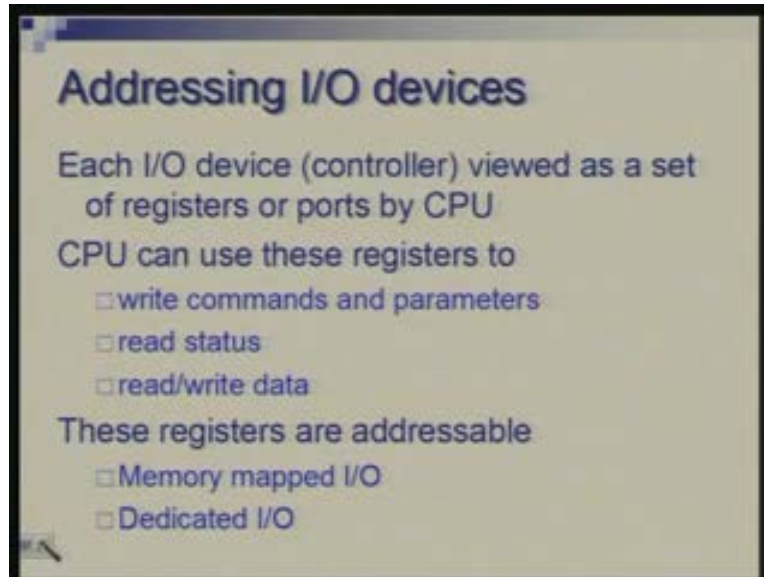
In earlier days the graphic display or the alpha numeric display used to be a peripheral separately which was linked through a serial port. It would mean that sequence of characters or sequence of bytes would be sent reflecting what you want to display or the change which you want to make and then some intelligence in the device would take that sequence of characters and modify the display accordingly. So there, the rate at which you can change things would be limited by the rate at which you can encode the changed information and send it over serial ports.

But now in PCs or in workstations the information is directly picked up from a buffer in the memory and display it so you can continuously refresh it there. Look at disk drive. In the disk drive typical operation you may like is that before you do the data transfer; you want the **head** read write head to be positioned on certain track and certain sector. So here is an operation which does not involve bulk of data transfer, it is a preparatory action which is required.

So now, looking at these four examples which I have taken, you can see large variety in terms of the rate at which data gets transferred, the quantum of data which is handled and the mechanism of data transfer. For example, is data automatically being picked up from some area or it is being explicitly transferred as and when required. The graphic display was an example where automatically the data keeps getting transferred whereas in network or in mouse or keyboard example we are doing it on demand. As some keys get depressed on the keyboard then the information gets transferred or in network, for example, when something comes from outside on the network then a transfer occurs.

In graphic display you may change that pixel matrix as and when you want but transfer from pixel matrix to the display device is a continuous activity. Now, for any of these operations you need to address..... there are large number of devices, you need to talk to them, the processor needs to address them before it could talk to them. So each device or strictly speaking, the controller of the device, appears as a set of registers to the processor. You can view them as registers or view them ports in the sense that they are actually allowing you to talk to the outside world so you can also call them ports.

(Refer Slide Time: 15:06)



So what do you do with these ports?

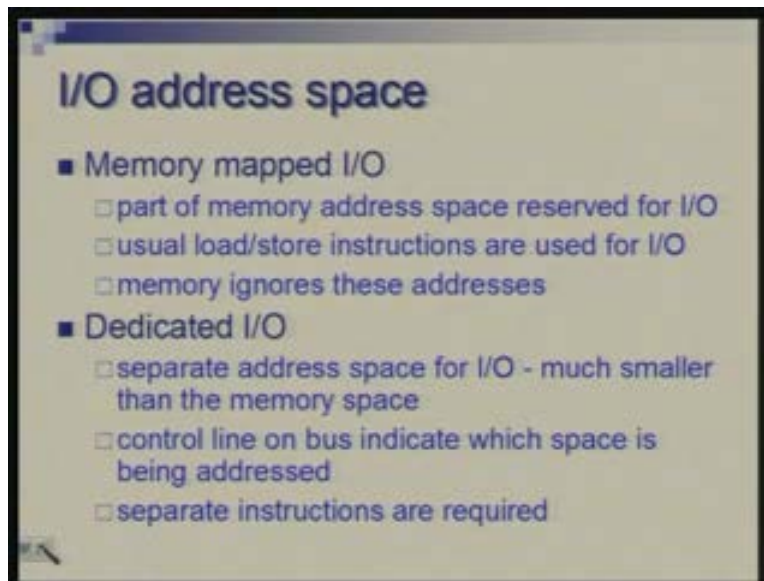
You can, for example, give commands to a peripheral device. So there would be some register in which commands could be given. So you load some code in a particular register, it will be treated as command and the device would be expected to work on that command. Command may also involve some parameters. For example, a command to seek the disk, you may have to specify the track number and sect number. You can read the status of the device. So basically certain register would be designated as status register and you can read that register and check specific bits to find out if certain things have occurred or not.

For example, if key has been depressed on the keyboard or a packet as come from the network and so on. Also, there could be registers through which actual data could be read or written. So **all these each** depending upon the requirement of the device each device would be seen through a collection of registers and these registers must have address so that at any given time a program running in CPU can read or write a register.

There are two ways addresses can be assigned to this: You can have a memory mapped I/O which means that in the entire **memory space** memory address space you can leave some area which is earmarked for input/output. So you might say, for example, at the higher end of the address space you are leaving 1k area which is not to be utilized for

memory and it is dedicated for I/O. So last one thousand addresses are reserved for register or it could be first few or whatever a particular architecture has in mind. So you are basically reducing the memory space by a small amount which may not really be very critical and that area is available for I/O. The other alternative is, have a separate address space for I/Os. So let us look at each of these.

(Refer Slide Time: 17:24)

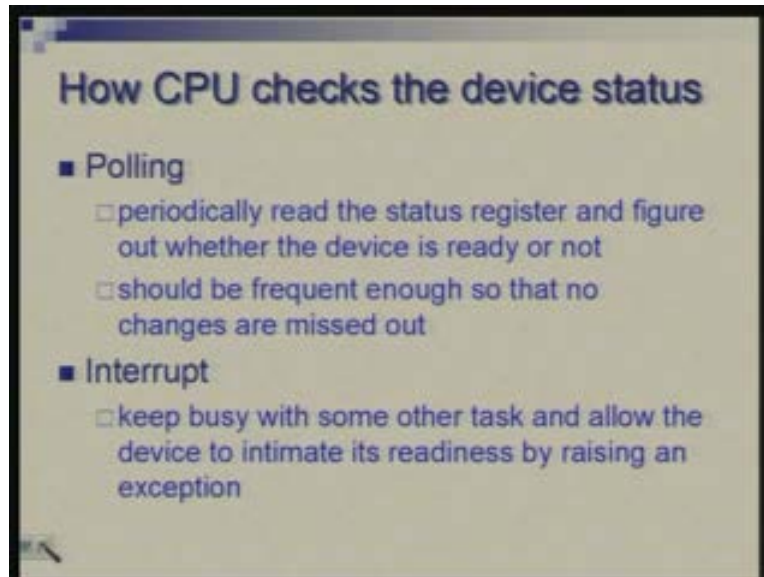


As I just mentioned, in memory mapped I/O, part of memory address space is reserved for I/O and you can use the usual load and store instructions for reading or writing the register which are actually located in that space. So, memory is designed to ignore the addresses which are following in that range. So let us say you are using last 1k addresses for I/O in a given address space. So **if a** if a load occurs, if a load instruction is executed and address is found to be following within that range, the memory does not respond. But appropriate I/O device to whom the address has been assigned will respond. So the data which would be sent to the CPU will not be by memory but it will be the particular I/O register which will send the data. And similarly, when you are doing a store instruction, depending upon where the address falls; address falls in memory range or I/O range, the data will go to memory or I/O accordingly.

In the second approach you have to have a different address space for input/output and it need not be as large as what you have for memory, it is much smaller, **it could be** you could have let us say 10 to 15 bits trying to address I/O devices, so it need not be a very large address space. And when processor is trying to read from memory or read from I/O device, the distinction would now be made by, not by the address value but by some control signals which will be flowing on the bus. So in the first case depending upon the address value you can determine whether address is meant for I/O or for memory. But in this case, same address could fall in both the spaces and therefore you need something additional to distinguish whether address is meant for I/O or memory. So there are additional control lines which are included in the bus and they indicate whether memory

is being addressed or I/O is being addressed and you need different instructions to work with I/O in such a case. So there could be additional instructions similar to load and store but different op-codes which will indicate that these are for transferring data to and from I/O devices.

(Refer Slide Time: 20:06)

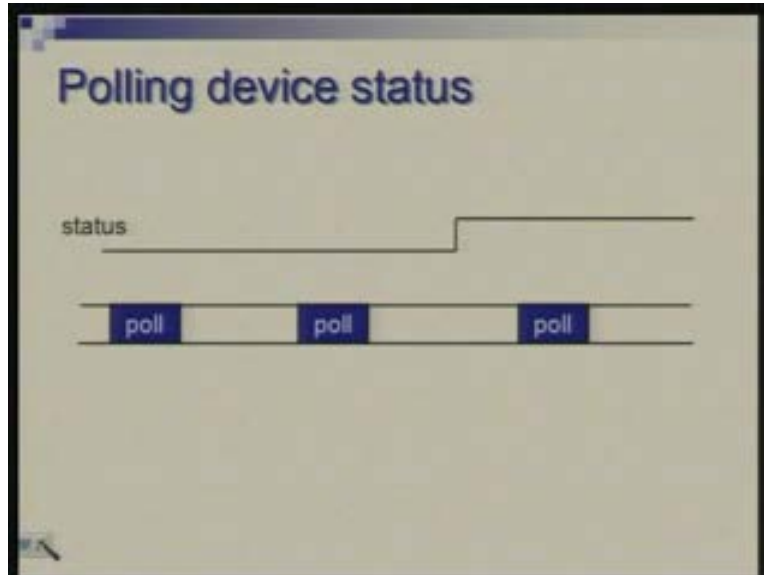


Before transferring data, a program would often need to check whether a device is ready for transferring data. So, for example, take the case of keyboard, the processor needs to determine if key has been depressed or not so it needs to periodically check whether key has been depressed and its code is available or not. This has to be done periodically and the frequency at which you sample this has to be enough so that you do not miss out anything. So if keys are being depressed at certain rate, let us say at the rate of 5 keys per second and you are checking only once in a second then you are going to miss out. So you have to check **frequent enough** frequently enough so that nothing is missed out; this approach is called polling where you are periodically sampling and **trying to see trying to** you would read a status register, check particular bit and figure out if what you are looking for is there or not.

An alternative is that processor does not get tied up in checking the status, it is busy with something else, it could be doing some other useful task and whenever there is a need for device to invoke action from CPU it will send an exception or interrupt. So the device will intimate its readiness by raising an exception. It basically means that a signal **which is going to** which is going as input to the processor is activated and processor would have a hardware which will transfer its control to exception routine wherever that signal gets activated.

In general, a processor may have multiple interrupt signals coming from outside devices.

(Refer Slide Time: 22:22)



In terms of time, you can visualize the polling activity like this that, let us say, some status is changing at this particular instant (Refer Slide Time: 22:32) and you are polling at certain rate; you polled here, spent some time in reading the status and figuring out if device is ready or not, you found it is not ready, do something else, come back, check it again, found it not ready, come back here, find that status is ready then you carry out the transfer.

(Refer Slide Time: 22:54)

Overheads of polling - example

- Processor clock : 500 MHz
- Cycles required for polling : 400
- Find overheads for
 1. mouse - polled at least 30 times a sec
 2. FDD - 50 KB/s, 16 bits at a time
 3. HDD - 4MB/s, 4 words at a time
- Answer
 1. $30 \times 400 / (500 \times 10^6) = .002\%$
 2. $\frac{1}{2} \times 50 \times 10^3 \times 400 / (500 \times 10^6) = 2\%$
 3. $(1/16) \times 4 \times 10^6 \times 400 / (500 \times 10^6) = 20\%$

Let us look at an example and put some numerical value. See what would be the implication of such an activity. Suppose there is a processor running at the rate of 500

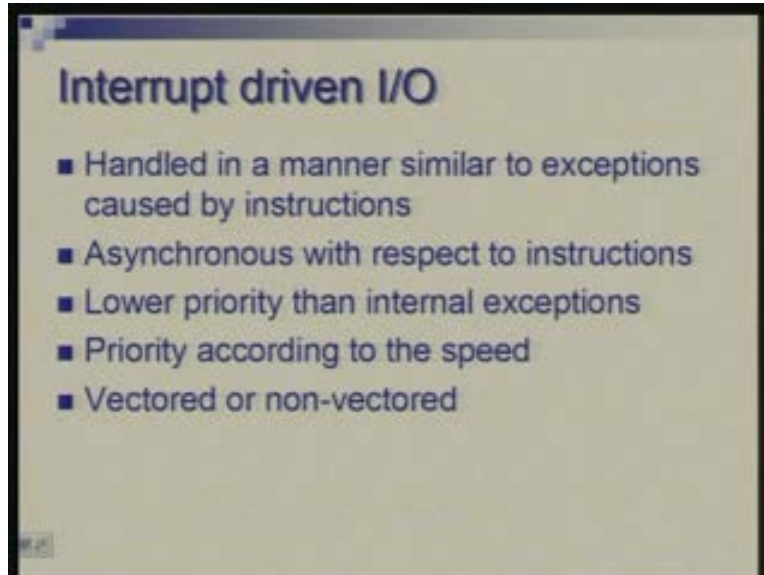
megahertz or 500 megahertz clock is there and every time you need to go and poll and take a decision; suppose it consumes 400 cycles then we want to see how much is the polling overhead for different types of devices with different speeds. So let us say first example is mouse which needs to be polled let us say thirty times a second and this figure is keeping in mind how fast one would typically move the mouse. Second case is; you have floppy disk drive which transfers the data at the rate of 50 kilobytes per second and each time every time you get 16 bits of data or 2 bytes. And thirdly, you have hard disk drive which transfers data at 4 megabytes per second and the unit of data **of data** which comes out is 4 words at a time. Now, in these three cases there is a vast variation in terms of speed and your polling has to be faster and faster accordingly.

As we will see that when you have to poll very frequently you are **you are** wasting lots of time and CPU gets tied up to a significant extent in this polling activity. So in the first case we are we are trying to poll at the rate of thirty times a second and every time we poll **we are wasting**..... **let me not say wasting**, we are consuming 400 cycles, so thirty times 400 is the number of cycles consumed in polling every second and CPU is running let us say 500 cycles in 1 microsecond or 500 million cycles per second. So, as a fraction or as a percentage what is the fraction of the cycles which actually are lost in this activity and not in executing other computation instructions. So we divide this, this is the number of cycles per second (Refer Slide Time: 25:28) divided by this cycle per second we get a ratio, convert that to percentage you get 0.002 percent so it is a very small fraction of the time which is spent and we will certainly not mind doing polling of mouse in this particular manner, there is very little overhead.

As you go to floppy disk drive the rate of sampling or rate of polling is higher so we have to poll at least at this rate. every data we should every **tie is** coming at this rate 16 bits at a time so basically we are getting 2 bytes at a time so number of times we are actually transferring is 50 divided by 2 into 10 raised to the power 3 so this is 50 kilobytes per second taken half of it because divided by 2, 2 bytes at a time so we must poll at least these many times per second and again similar factors 400 cycles lost every time you poll divided by the processor clock we have 2 percent that means only 2 percent of the cycles are dedicated for this so this is not as comfortable as mouse but still tolerable. The only problem is if we had several of such devices then these percentages would add up and we would have lost sufficient time. But this is an isolation but still a tolerable device if you do not have too much of other activities.

Finally we move to an even faster device 4 megabytes per second and 4 words at a time. So this 4 megabyte per second figure is divided by 16 because we are trying to transfer 4 words or 16 bytes at a time and each time we transfer we spend 400 cycles divides by the clock rate we get a figure of 20 percent which is which is clearly undesirable; you cannot spend 20 percent of the time just polling a particular device so we need to do something better than that and solution for that is interrupt driven input/output.

(Refer Slide Time: 28:13)

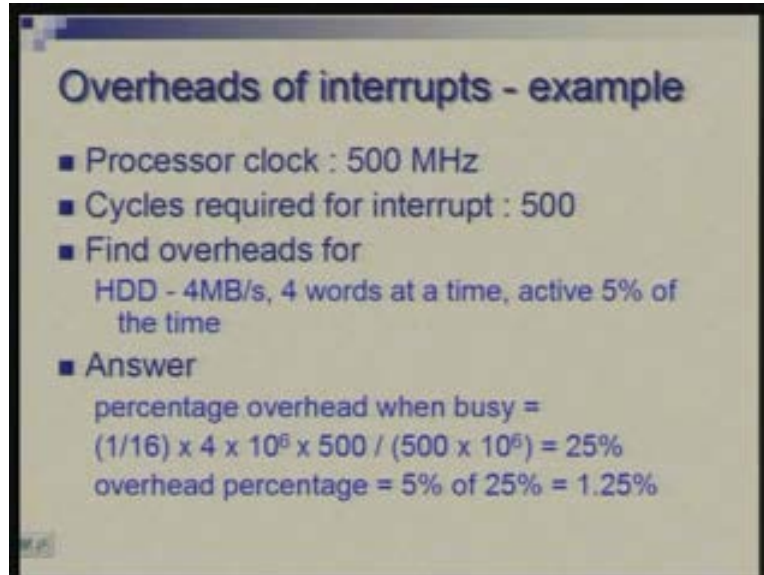


Here what we do is we allow a device to inform the processor about its readiness. So processor will probably instruct the device and ask it to send an interrupt or send an exception as and when it needs processor attention. So processor attention is given to the device by transferring control to exception handling routine and exception handling routine would do the needful. The way you handle exceptions here coming from external word of input/output, the mechanism is same as what you for internal exceptions which may come because of arithmetic overthrow or illegal instruction or page fault and so on.

The difference here is that these exceptions which are coming from outside they are asynchronous with respect to the instruction whereas internal exceptions will come at specific time. For example, arithmetic overflow would be detected in a specific cycle of an instruction. But exception or interrupt from I/O device could **come at** come in any cycle; it has no links with a specific instruction. Therefore what you would typically do is that, at the end of instruction execution, you will see if there is an exception from external world and then respond to that. These exceptions are given comparatively lower priority as compared to internal exceptions. Among these external exceptions the priority may be according to the speed. So devices which are faster need early attention and therefore there are given priority, slower devices are given lower priority so you might have several exceptions occurring from I/O devices at the same time and you will serve them according to the priority.

Similar to internal exceptions which we discussed earlier the mechanism could be vector interrupt or non-vector interrupt. Vector interrupt means that the control directly gets transferred to an exception handler which is meant for a specific exception and in non-vector case you set the calls of exception in a particular register but transfer of control takes place to a specific location then you are going to run some code, check that call register and branch of to one of the points, so both the possibilities exist here also.

(Refer Slide Time: 30:53)



Overheads of interrupts - example

- Processor clock : 500 MHz
- Cycles required for interrupt : 500
- Find overheads for
HDD - 4MB/s, 4 words at a time, active 5% of the time
- Answer
percentage overhead when busy =
 $(1/16) \times 4 \times 10^6 \times 500 / (500 \times 10^6) = 25\%$
overhead percentage = 5% of 25% = 1.25%

Now let us get back to a similar scenario and try to quantify the overheads of interrupt driven transfer. We have same processor working at 500 megahertz. Now let us say that overhead per interrupt is little more than the overhead of polling, suppose it is 500 cycles. And, we want to see how it is going to work in case of hard disk drive.

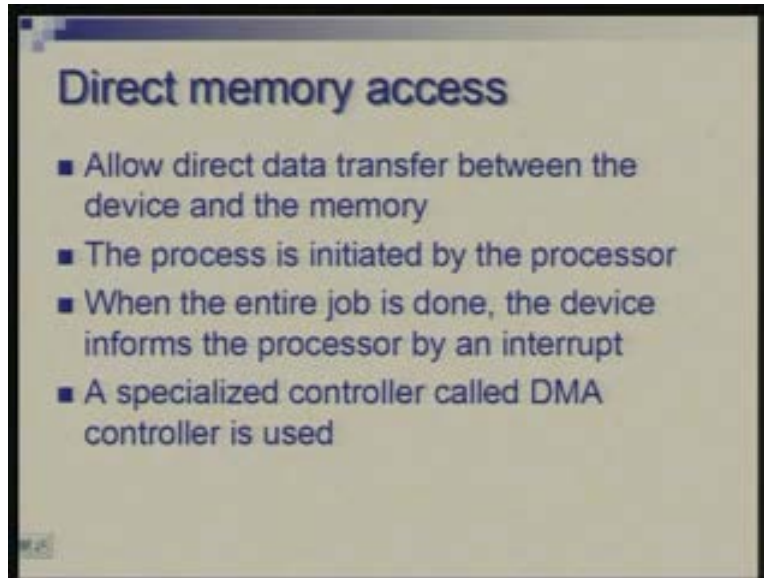
Now, if disk drive was always active then things are going to be as bad as polling; you will get continuous interrupt. In fact it would be worse because every time there is an interrupt the overhead is larger. But the reality is that device or hard disk drive in this particular case will not necessarily be always active. So suppose it is active in entire duration of program execution, only 5 percent of the time, when it is busy data is coming at a fast rate and when it is not busy there is no activity. So in such a case we will definitely have saving of time here because the device would inform only when it needs the attention. When it is inactive nothing will happen and there are no interrupts. But in polling based approach you have to continuously keep polling.

Now we have the overhead if the device was continuously busy we have same expression, 4 megabytes per second or 1 by 16 because we are transferring 16 bytes at a time, this is the overhead (Refer Slide Time: 32:39) number of cycles per interrupt and divided by the clock frequency you get 25 percent, so instead of 20 percent we have 25 percent but since we are assuming that the device is only busy or active, 5 percent of the time, we take 5 percent of this and the real overhead will be 1.25 percent so that is where it will score.

Now at times, even 1.25 percent overhead may also not be desirable particularly when you have number of such devices, hard disk drive in a system is only 1, you have display, you have network and you have other things. So, if all such overheads are added then lot of process time will be consumed. So a solution for that is not to involve processor in transfer of every word or every group of words. What you could do is you could allow

the device to deposit data directly into the memory or read data directly from the memory and that operation is called direct memory access.

(Refer Slide Time: 33:52)



So the processor's role in direct memory access would be only to initiate the transfer or set up things to tell the device that this is the amount of data you need to get. Let us say if it is disk, it is track number so and so, sect number so and so, get so much of data and it also specifies that this data has to go to this particular area in the memory. So once that initialization is done the processor can get busy with something else and as the words or bytes come from disk drive they get transferred directly into the memory and processor does not come to know of that. The processor of course needs to be informed when the entire job is done and at that time processor can **see** check the transfer has taken place correctly or not.

For example, if in between you found a bad sector or something then some error occurred then status would be set accordingly and the processor interrupted. So the processor has to see at the end whether the I/O operation ended successfully or there was an error. This entire job actually is entrusted to special controller which is a special piece of hardware, it is called DMA controller. So it is in this DMA controller you would specify memory addresses, amount of word, amount of data to be transferred, the direction of transfer and what is to be done at the end of the transfer. The DMA controller could actually take care of not just one peripheral device but multiple peripheral devices which are required to transfer data directly to the memory. So it could be a 2-channel DMA or 4-channel DMA, 4-channel DMA, for example, would allow four high speed devices to transfer data to memory or read data from memory. So basically you could say it is an extension of the processor logic or augmentation of the processor logic which looks after this.

So the sequential address in the memory will be provided by this DMA controller. It will start from the starting address and as and when data comes it will keep on updating the memory addresses and detect when the whole transfer is done.

(Refer Slide Time: 36:22)

Overheads of DMA - example

- Processor clock : 500 MHz
- Cycles required for initiating DMA : 1000
- Cycles required for interrupt : 500
- Block size : 8 KB
- Find overheads for
HDD - 4MB/s, always active
- Answer
percentage overhead =

$$\frac{[4 \times 10^6 / (8 \times 10^3)] \times [(1000+500)]}{(500 \times 10^6)} = 500 \times 3 \times 10^{-6} = 0.15\%$$

Now let us try to quantify the overheads in a DMA like situation. We again have same processor running at 500 megahertz. Suppose initiation of DMA requires 1000 cycles, so everything initially is setup and that is one time when processor gets involved. Next, the processor gets involved when end of DMA interrupt comes and let us say 500 cycles get consumed in that. So **in the in the** in one DMA activity basically the CPU is spending 1000 cycles plus 500 cycles. Let us imagine that the block size or the chunk of data which gets transferred in one DMA is 8 kilobytes. So, by spending this 1500 cycles the processor will achieve transfer of 8 kilobytes and for the next 8 kilobytes you may have to repeat the process. So we can now even look at situation when hard disk drive of the kind we discussed in previous example is always active.

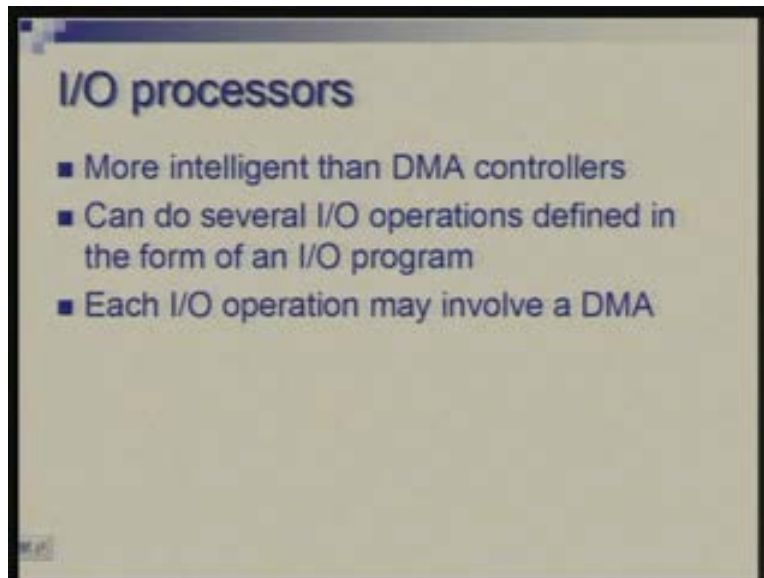
Suppose it needs to transfer 4 megabytes per second continuously and the overhead then will be.....you have 4 megabytes per second transfer rate and since **each time you are** each DMA operation requires 8 kilobytes (Refer Slide Time: 38:00) the ratio of these two will give how many DMAs per second are being done and each time you do DMA, you are spending these many cycles; this in the beginning and this in the end divided by the clock frequency will give you the fraction of time spent on the DMA by the processor.

So of course DMA controller has to do lots of work. The work has been taken off or outsourced by the processor here. So this comes out to be..... (Refer Slide Time: 38:34) you can see that the first factor, the ratio of these two is 500, this is 3 into 10 raised to the power minus 6 so it comes out to be 0.15 percent which is quite an acceptable figure.

As we have seen from polled transfer to interrupt transfer to DMA transfer, depending upon requirement of the device, we can choose a suitable mechanism for transfer and as you go along this progression there is higher and higher performance but there is a cost issue also. DMA controller means additional cost. Similarly, interrupt also requires certain mechanism so there is extra hardware which also adds to the cost.

One step further from DMA is the concept of I/O processor. A DMA can be initialized to do one block transfer but an I/O processor could actually be asked to do a sequence of many DMA like operations.

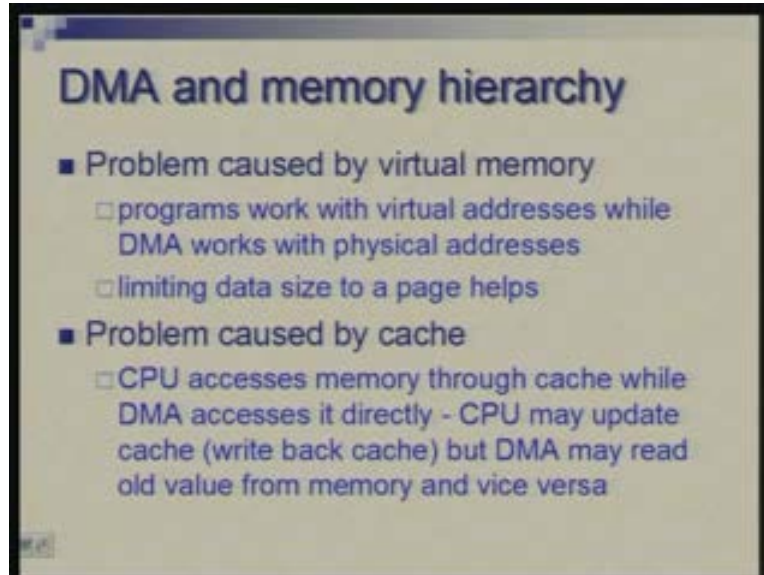
(Refer Slide Time: 39:47)



So it could be asked to do, let us say, read a block from the disk that is operation number 1, send a block of different size from a different area in the memory to the network controller that could be second operation, third could be send a block of data to a printer and so on. So a sequence of operations each typically involving a DMA of certain block size could be defined and I/O processor could go through this sequence. So, for an I/O processor each appears like an instruction or an I/O operation. These instructions could be stored as a small I/O program somewhere either in a dedicated memory for the I/O processor or somewhere in the main memory. Each I/O operation is a major activity and I/O processor will sequence from one to another in this particular manner.

Now when we are talking of DMA essentially there is a direct communication between memory and one or more I/O devices. Now how do you view this in light of the fact that a memory is not something flat, there is a memory hierarchy. You could have cache one or more levels and you also have virtual memory, so how does this whole thing work when memory hierarchy is present. So what are the problems caused by presence of virtual memory?

(Refer Slide Time: 41:20)



The programs which are running in the processor, they look upon memory in terms of virtual addresses whereas the DMA devices will typically work with physical addresses so there is a discrepancy between these. What it may mean is that you would need to do translation of the address so when you are initiating a DMA you would need to specify a physical address for the DMA controller. So you need to tell DMA controller that transfer, let us say, 8k bytes of data to this physical address and the program would be working in in terms of virtual addresses. So, if you have created an array you know its virtual address, so there has to be an explicit conversion of that virtual address to get to the physical address.

Now, remember that the virtual addresses do not map linearly to the physical address. So, in virtual address you have, let us say, one page and then next page contiguous but they may map to two pages in the memory which are not contiguous which may be at arbitrary addresses. So it may not be sufficient for this translation to be done once.

One of the **one of the** ways to handle this is that you restrict each DMA activity to utmost a page. Suppose a page size is 4 kilobytes and you want to transfer 8 kilobytes then there is a problem that **for** your data is coming in two pages, you need to specify the physical address for each page, you cannot specify one physical address and say that transfer 8 kilobytes of data to this because it may not go to contiguous areas in the physical pages. You want them in contiguous area in virtual space but in physical space it is not contiguous.

If you **are limiting if you** limit your data size or DMA's data size to one page then this problem is taken care of. So basically if you want to transfer larger piece of data you have to break it up and incur the overhead of multiple DMA's.

What about cache? When you have cache what problems it can possibly bring in?

There is essentially a problem of consistency because CPU is directly talking to cache and not to the memory whereas DMA device will typically have no access to the cache, it write to the main memory or read from the main memory so there is a problem of consistency. One of the two memories may have more recent information as compared to the other. So for example if there is a DMA input happening, the information coming from devices **getting put** in the memory and suppose it **gets put** in the block for which copy exists in the cache then what you have in the cache is out of date; the information in the main memory has been updated by the device and what you are seeing in the cache, what processor you are seeing in the cache is stale information, it is not up to data.

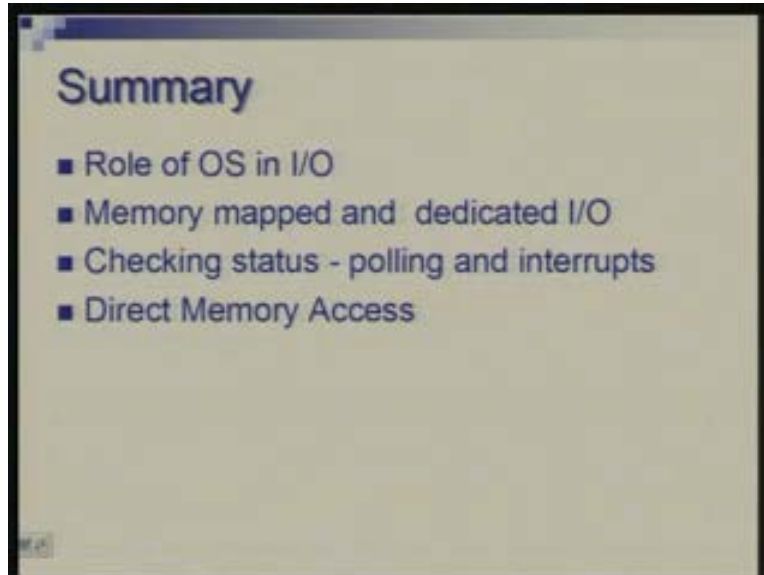
You can also have reverse happening that CPU has put something in a cache block and it is expected to be sent out to a device through DMA. But DMA control will pick up information from memory which is not yet updated particularly with write back cache where **there is a** you are not updating main memory immediately. So in write back cache, if you recall, data in the memory gets updated only when a block is getting evicted. So you keep on making updates into cache blocks and but if corresponding block from main memory was being returned to a DMA device then you have a problem. Actually similar problem occurs in multiprocessor cases also.

Imagine that you have multiple processors which are trying to share same memory; they would typically have their own caches. So just imagine two processors they own cache but a common shared memory. So once again if there is a data which processor 1 has to communicate to processor 2 through shared memory the two caches may be inconsistent. So the solution for this lies in actually what is called cache **coheres** protocol.

[Conversation between Student and Professor: (46:42)] you are talking of the first problem, virtual memory. Yeah, if you are limiting your data size to a page for DMA transfer then you are definitely increasing the overhead. Suppose you want to transfer 16k bytes of data in DMA and your page size is 4 kilobytes, if this virtual memory problem did not exist you could have incurred the DMA overhead only once. But now you will have to break it up into four DMAs each time you supply appropriate physical address the DMA occurs without any problem but the overhead has occurred four times, so you increase the overhead.

The problem of cache in coherence is solved by what is called cache coherence protocol. These protocols are additional actions which cache controllers have to take when multiple copies of data exist in multiple caches and memory. So there could be one or more cache in memory which is the cause of having multiple copies of data and something is getting updated something is not getting updated which is the problem of coherence. So these protocols (**I will not go into details of this**) are designed so that coherence is maintained. At least if two copies are going out of sync, everyone comes to know that they are out of sync and one way still the updation occurs. So this also means additional overhead, additional work which is to be done. I will stop at this point and summarize.

(Refer Slide Time: 48:30)



We started by looking at the role of system software in I/O operation and we noticed that software is required, a system software particularly has to handle device specific details, it has to handle interrupts, it has to take care of sharing of device by multiple processor, look at the file organization and so on. We saw that devices are viewed as set of registers by the processor and they can be addressed either by using part of the memory address space or defining separate address space. We saw two mechanisms of checking the device status: polling and interrupts. Polling has higher overhead, particularly it becomes undesirable for fast devices and interrupts is used. Then there are times when interrupts are also not acceptable, you have to have direct communication between memory and the device that is called direct memory access which is which is a complex mechanism but it is essential for fast devices like network controller, disk drive and so on. I will stop at that, thank you.