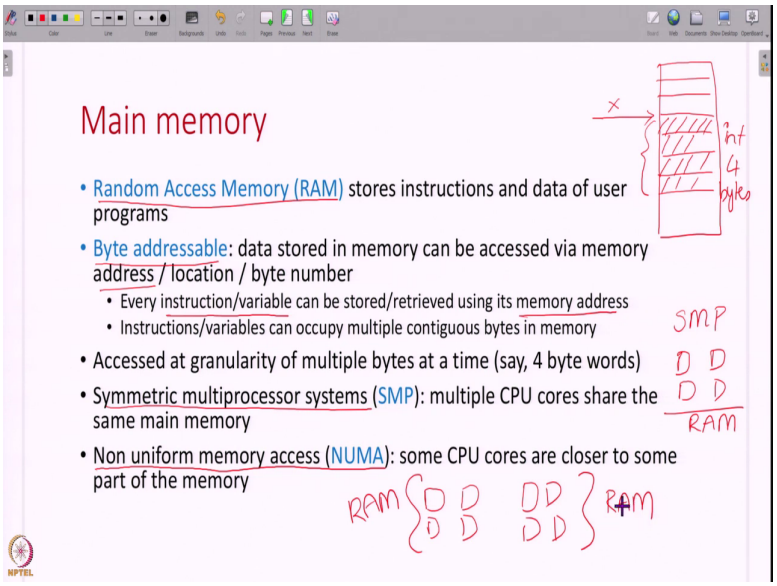
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture - 4
Overview of memory and I/O hardware

Hello everyone, welcome to the fourth lecture in the course, Design and Engineering of Computer Systems. So, in this lecture, we are going to continue our discussion of the hardware, which is the building block for modern computer systems. In the previous lecture, we studied how CPU hardware works. And in this lecture, we are going to study how the memory hardware works as well as how IO devices work. So, let us begin.

(Refer Slide Time: 0:41)



So, the main memory is also called Random Access Memory, because you can access any byte of the memory by providing a suitable address. So, you can think of your memory as an array of bytes. And each byte stores some data and the memory is byte addressable, that is you give a certain address you say, give me the data at address x, and this data at address x is returned to you by the memory hardware.

So, now, how this memory hardware works, you have to take a course on computer organization and architecture to understand the hardware details, but for the purpose of computer systems for a high level view that is required, it is enough to know that memory is byte addressable and every instruction or a piece of data or variable that you store in memory can be retrieved using its memory address, that is you give the byte number you will get back the data that is stored there.

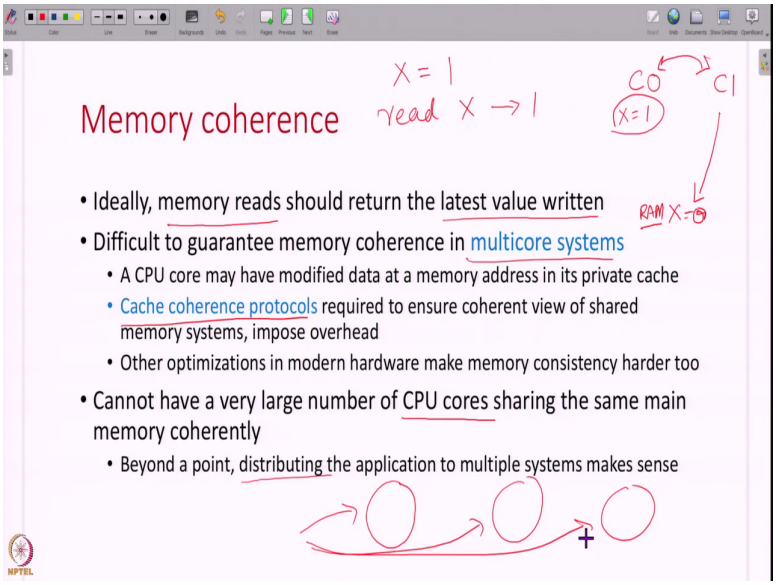
So, note that every instruction variable need not just occupy one byte. So, suppose you store an integer in memory, it will occupy four bytes, an integer is four bytes in size. So, when you access an integer at address x, the next four bytes will be given back to you. So, memory is accessed not just at byte granularity, you can address it at the granularity of bytes, but you can access it at a granularity of multiple bytes, say four bytes at a time eight bytes at a time, this is also called a memory word.

So, moving on, now, there are multiple CPU cores, how do they share memory? Normally, most systems are what are called symmetric multiprocessor systems that is you have multiple CPU cores, and all of which access the RAM uniformly. All cores share the same main memory have the same view of main memory uniformly. But, recently, you are also having what are called NUMA machines, NUMA stands for Non-Uniform Memory Access, whereas SMP stands for Symmetric Multiprocessor Systems.

So, with non-uniform memory accesses it so turns out that for some CPU cores, some memory is closer and for other CPU cores, there is another memory that is closer. So, while any core can access all the RAM, some memory addresses are

closer to some CPU cores and some are closer to the other CPU cores. So, these are called NUMA machines.

(Refer Slide Time: 3:27)



So, whatever it is, whether it is SMP or NUMA, we have this property of memory coherence, that is, all CPU cores will see and be able to access all the memory addresses coherently. So, what do I mean by coherently? What we mean is that when you write some value, say at memory address x, I write a value of one, then when I read back the value of return at address x then I should get back a value of 1.

So, whatever I have written the latest value return that should be returned in the memory reads. So, this is the basic expectation of a memory system. This is called memory coherence. But of course, this might look like obvious, if I write one, why will not I get back one you might be wondering, but this is not so easy to guarantee in multi core systems.

So, suppose we have seen this in the previous lecture that some CPU caches are actually private to a CPU core. So, if you have core 0 and core 1 in its private cache, it has returned the value x equals one return the value of one at some memory location x, then if core 1 tries to go to RAM, in RAM, the value might

still be the older value. The RAM may not have been updated from this private value.

So, it is possible that this core will read a wrong value and not the latest written value. So, to avoid this, we have seen in the previous lecture that there are these cache coherence protocols that will run. These two cores will talk to each other. And this core will realize that the latest values here, and it will get that value and so on.

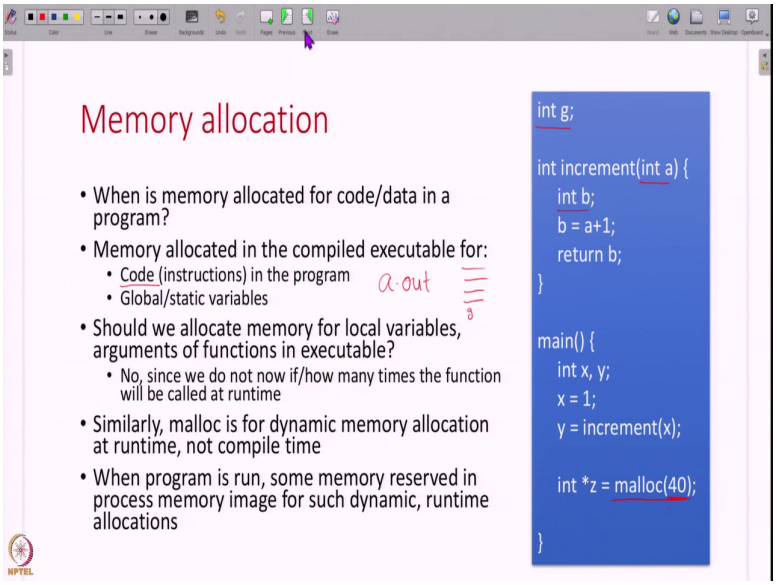
But all of this imposes a certain amount of overhead on the system, these cache coherence protocols. There are also other optimizations in modern hardware that make this memory consistency and coherence a little bit harder to achieve. Therefore, while this is being done in modern systems, this imposes a certain limit on the number of CPU cores that can share the same main memory coherently, which is why you do not have millions of CPU cores accessing the same memory.

Because if you have millions of cores like this, all of them running their cache coherence protocols, it might get very-very complicated after a point of time. Therefore, there is a limit in practice, about how many CPU cores can share the same main memory, same set of bytes coherently.

And if your number of CPU cores exceeds that point, if your application if your computer system needs more than the number of cores that are in this practical limit, then you will basically have to distribute your application. You will have to just run your application on multiple different replicas and split the traffic. So, this is where the concept of distributed systems comes in.

Beyond a point certain number of cores cannot simply coherently share the memory and then you say let me distribute my application into multiple different systems each having their own CPU cores and memory.

(Refer Slide Time: 6:32)



So now, let us understand how is memory allocated for the code and data in a program. You as a user have written a program say this is a simple example of a program, there is a main function. It has some variables, and there is another function defined the increment function, which basically increments the arguments sent to it and returns it. And this main function calls this increment function. So, this is a simple program.

So, now the question comes up, how do we allocate memory for the code and data in this program? So, the memory for the code, the instructions in this program is allocated when you compile the program, when you compile the program, the C compiler, if this is simple C code, the C compiler will translate this program into instructions that the CPU can understand and these instructions are stored in your executable.

And also, certain global variables like this variable g over here, you know that you will always use it somewhere or the other in your program. Therefore, for that also the memory is allocated at compile time, when you create your

executable itself, your a.out will have some memory laid out for all the instructions as well as for your variable g.

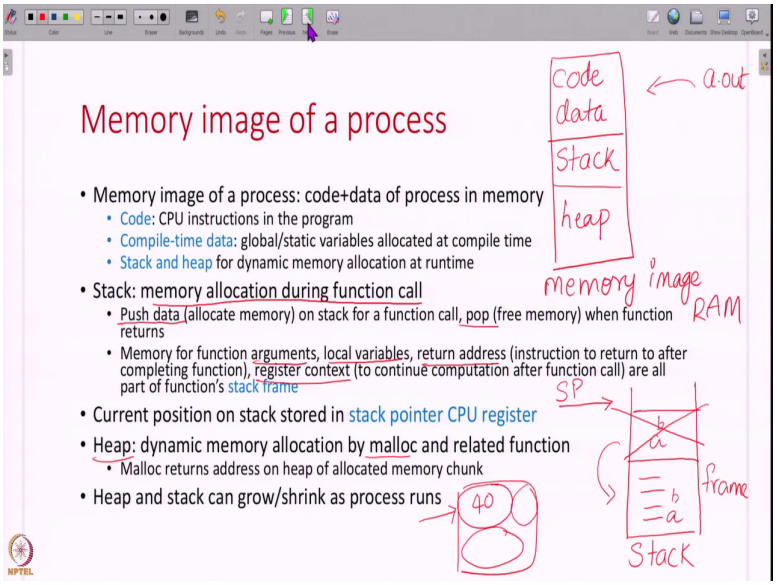
Now, what about variables in a function? In this function increment, there are arguments, there are local variables, when should you allocate memory for these functions. So, note that you cannot allocate memory for these functions in the executable itself, why? Because you do not know if this function will ever be called in your program.

And even if it is called, you do not know how many times this function will be called, what of this function increment is called twenty times, then you need twenty copies of variables a and b, if it is only called ten times you need only ten copies of these variables. So, how many copies of these variables will you allocate in memory? You cannot, it is hard to know.

Similarly, when there are calls like malloc, if you have studied dynamic memory allocation in a programming course, malloc is a way to allocate 40 bytes. Here in this example, dynamically, now, again, this number 40 may not be known to you at compile time. So, you cannot allocate memory requested by malloc in your executable also, when you compile your code.

So, for certain variables, and for the code in your program, you can allocate memory at compile time in your executable, but for certain things like function arguments or malloc dynamic memory allocations, you cannot allocate memory in your executable, memory has to be allocated when the program is running.

(Refer Slide Time: 9:17)



So, this with this background in mind, let us understand how the memory image of a process looks like. So, recall that when you compile your program, there is this a.out an executable created on disk, then you will load this into memory to create a memory image of a process in RAM.

So, memory image is the memory allocated to a process in RAM. So, this memory image has multiple parts, you will have the code and you will have compile time data like global variables, static variables, which are allocated at compile time itself, because you know there is only one copy of the data that will be used you will allocate it when you create the executable itself.

On the other hand, you will have separate sections in the memory image called the stack and the heap for dynamic memory allocations. So, what is the stack? The stack is used for any memory allocation needed during a function call. So, when you make a function call, that is when you will need to allocate memory for the arguments, for the local variables and a few other things that I will come to in a little bit.

So, there is a separate area of memory in your memory image called the stack and the way the stack works is whenever you make a function, you will push an area of memory on the stack you will reserve an area of memory on the stack called the stack frame, and in the stack you will allocate all the memory for the local variables in the previous function a, b all of that will be allocated here. And if this function is called again, once again, you will push another frame, allocate memory again for the function for the variables a, b and so on.

And when you return from a function, you will pop the stack frame and go back. So, this is our stack, data structure, you must have seen a stack data structure in a previous course. So, this is how the stack is used for memory allocation during function calls. You will push data that is needed for the function call on the stack. And when the function returns you will pop or you will free up that memory after the function is completed.

And this is if a function is called multiple times, you will push multiple stack frames, allocate multiple copies of the variables on the stack. So, this allows you dynamic memory allocation at run time. And what are the other things you push onto a stack in addition to the arguments, local variables, there are also a few other things like return address where should a function return to that information also has to be stored somewhere that is also stored in the stack.

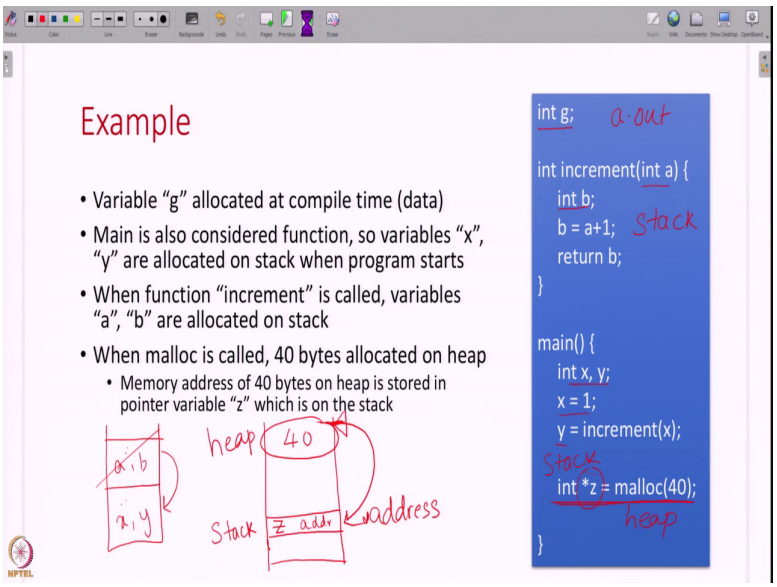
And also, the register context. If you are in the middle of a calculation, you have loaded some values into CPU registers. And then you make a function call. You have to save those values and CPU registers. So, that when the function returns, you can continue whatever you were doing. Therefore, various other pieces of information that are needed to safely return from a function call. All of these are also stored in the stack as part of a functions stack frame.

So, where do you know what part of the stack you are using, how much you have pushed, where you have to pop. So, which is why you have this special CPU register called the stack pointer register, like the program counters, this also stores the address of the current position that you are in the stack and more function calls you will keep pushing more and more into the stack, the stack pointer will keep on moving as you return from functions, the stack call stack pointer will keep going down like this. So, this is the stack in the memory image.

Similarly, you have another area in the memory image called the heap where malloc kind of memory is allocated. So, the heap has a lot of memory. And if malloc says give me 40 bytes, then this 40 byte chunk is returned by malloc. And if my lock says give me 80 bytes another chunk of memory, another chunk of memory is allocated from this heap.

And the heap and stack can grow and shrink as the process runs. If you are making many function calls, you push a lot of things onto your stack. Your stack goes up as you return from functions that goes down, as you do more malloc your heap expands, heap contracts. So, these two parts of the memory image are typically dynamic in size and their size can keep on changing.

(Refer Slide Time: 13:48)



So, going back to our example, we have a variable g here this is allocated, where is the memory for g allocated? This is allocated in your executable itself. On the other hand, variables like a, b in your function, all of these are allocated on the stack when the function is called in the stack frame. Similarly, the main is also a function, main is also not any special code it is also a function. So, when your program starts the main function starts.

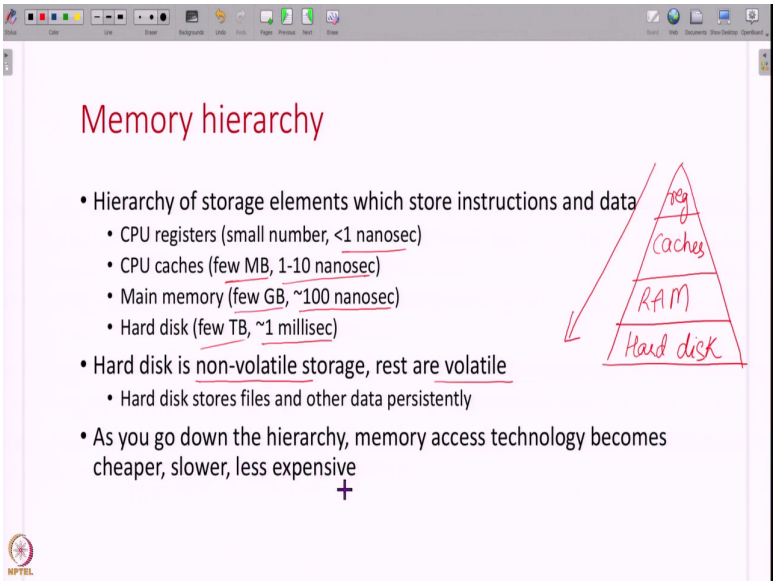
So, the function, the memory for these variables in main is also allocated on the stack. So, your stack will have initially main is a function called so x, y and a few other things then you call the function increment then you will have a, b and a few other things. And then when this function, incremental returns your stack, once again, all of this memory is erased. And your stack will once again have x and y, so this is how the stack grows and shrinks during a function call.

So, this is another interesting piece of code. There is a, you have done malloc, you have gotten a pointer and this is a pointer to an integer. So, where is the memory for this variable z allocated? Recall that z is a variable in the stack. Therefore, the variable z itself it is part of main, therefore, variable z itself will be in the stack, but this malloc 40 bytes of memory that is malloc. This will be

in the heap and the address of this 40 byte chunk that address will be stored in the stack in the variable z.

So, this variable z is a pointer. It is a pointer to a 40 byte chunk. So, the pointer address, this address will be stored in the stack, but the actual 40 bytes will be allocated on the heap. So, it is very important for you to understand what is the heap, these 40 bytes on the heap but this pointer is on the stack. So, understand the difference between the stack and the heap.

(Refer Slide Time: 16:07)



So, now, we have covered the various parts of what is called the memory hierarchy, the various storage elements that are there in a computer system. So, this is a hierarchy at the top of the hierarchy you have registers. So, whenever the CPU has to do any calculation, it will have to have that data in CPU registers which are like temporary storage, then where do the registers get their value from? They will get their value from CPU caches.

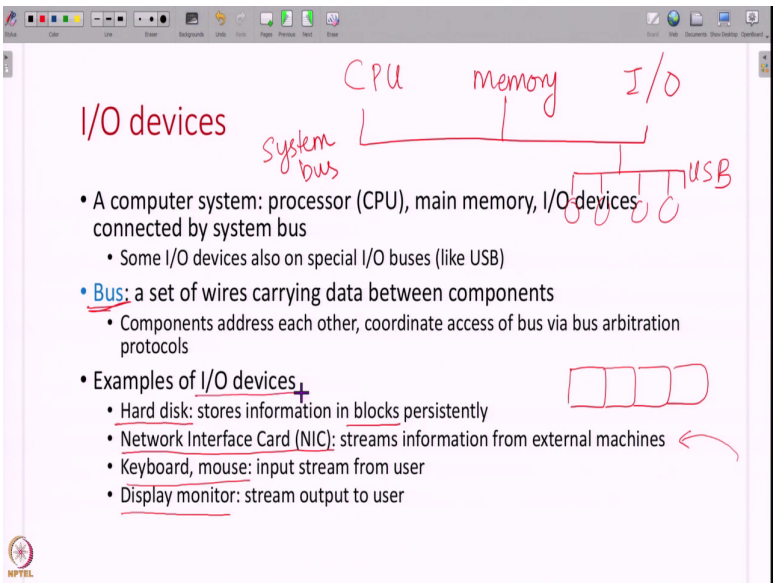
So, CPU will first, if it has to load some memory address into a register it will first check is it there in the cache. And this cache is also can be multiple levels, there is a hierarchy of caches. And if the data is not there in the caches, then you will go into main memory DRAM and you will get the data. And finally, below this DRAM is the hard disk or persistent storage where you store files. So, as you go down the hierarchy, the access speed, the latency is increasing.

You can access registers in under a nanosecond, caches take a few nanoseconds. Memory accessing DRAM takes few hundreds of nanoseconds, a hard disk might take a few milliseconds. As you go down this hierarchy the latency the access latency increases because the technology is different.

But of course, here the access latency is low then if registers can be accessed quickly why do not we just have everything as registers or caches because this is also more expensive, therefore, you only have very little registers, very little caches, your caches are only few MB. Registers are only a handful of registers in the CPU, but main memory can be few gigabytes, hard disk can be terabytes. Because it is cheaper, you will put more and more of it in your computer system.

And also, all of these three registers, caches and main memory are volatile, that is when you turn off your machine, the data in them disappears, whereas hard disk is nonvolatile storage. So, therefore, if you want to store anything persistently, permanently then you will put it in files and store it on the hard disk. And as you go down the hierarchy, access technology becomes such that it becomes cheaper, slower and less expensive.

(Refer Slide Time: 18:33)



So, now we have completed an overview of CPU and memory hardware. Now, let us understand how I/O devices work. So, if you look at any computer system, you will have a CPU, then you will have some main memory or RAM

then you will have a bunch of I/O devices. And all of these are connected by the system bus, what is called a system bus.

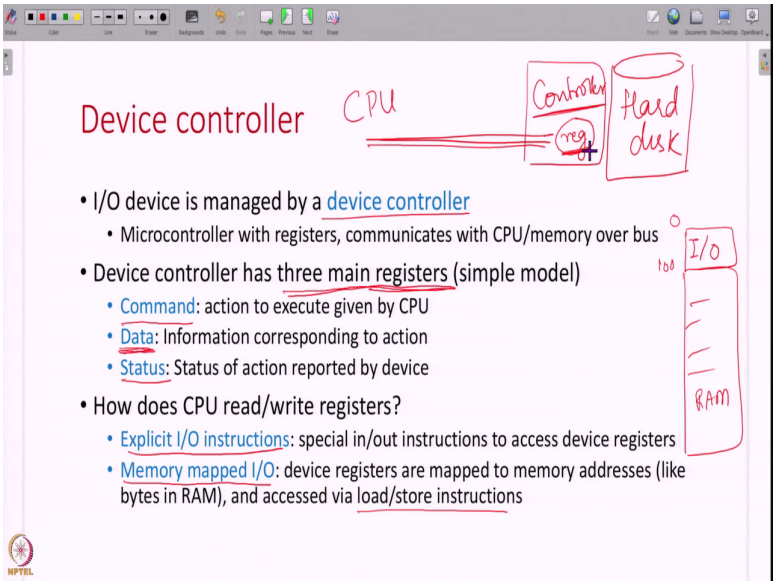
And of course, these I/O devices. There can also be other specialized I/O buses for various I/O devices as well. For example, there could be a USB bus and so on, where there are other more I/O devices connected. So, what is a bus? A bus is nothing but a set of wires that are carrying data between these components, the CPU will send some data to memory via the bus, the CPU will access I/O devices via the bus. These are just a set of wires.

And each of these components connected to a bus will have a certain address. And there will be some protocol for arbitrating access, who will write on, you do not want to people writing sending signals on the wires at the same time and gobbling up. So, there will be some arbitration protocols as to who accesses this bus at any given time and so on.

And you have various I/O devices that are connected, in this way. For example, you might have a hard disk that stores blocks. So, hard disks today store data in blocks of typically say 512 bytes and there are multiple such blocks in a hard disk and this data is stored persistently. This is a block storage.

Then you might have a network interface card that will stream information whenever another machine sends you some information that will come into your network interface card. Then you will have keyboard, mouse which are you know streams have input from the user, you have a monitor which streams output whatever the user is generating that output will be streamed to the monitor. So, these are all examples of I/O devices that you all must be familiar with.

(Refer Slide Time: 20:32)



So, every I/O device is managed by a device controller. So, the device controller is nothing but a small CPU like thing a micro controller, which has its own registers. So, for example, there is the actual hard disk which actually stores the data. And then there is a controller that is controlling this hard disk.

So, what this controller will do is, it will talk to the CPU or memory, it has some temporary storage in the form of registers, the CPU will give commands to it by writing into these registers, then the controller will read these commands executed on the hard dist send a response back, this controller is sort of the coordinator that is managing this entire hard disk.

And the way these devices communicate with the CPU and other components in a computer system is via registers. So, this is different from CPU registers, but it is the same concept, which is some temporary storage is there in the controller for communication. So, every device, every I/O device, typically exposes these three main registers.

Of course, there could be more or less depending on the device, but this is a very simple model that most devices follow. So, what are these three registers,

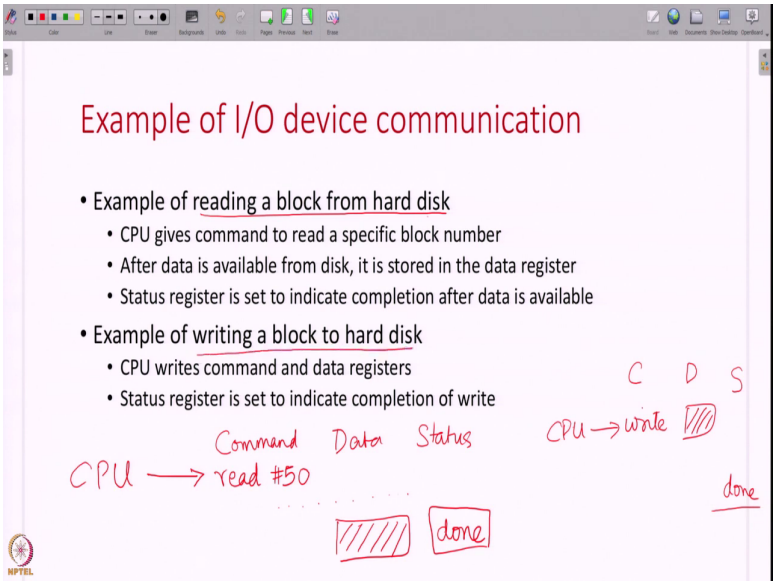
there is a command register, there is a data register and there is a status register. So, whenever the CPU wants to give any command to an I/O device, that command information will be written into the command register.

Whenever some data has to be exchanged, for example, data in a file, or data that has come over the network has to be exchanged, that is put in the data register. And when the device has to indicate its status saying I am busy, I am free, your command has been completed whatever status it has to indicate it is stored in the status registers. And the CPU will read and write these registers in order to communicate with the I/O device.

So, how does the CPU access these registers? There are two ways, you can either have some special I/O instructions, write special CPU instructions or explicit purpose is to write these registers that is called explicit I/O or you can have memory mapped I/O that is in your RAM, there are many bytes in the RAM. So, some of those bytes you can assign to I/O register say byte number 0 to 100, you say these bytes actually map to I/O registers and the remaining bytes map to RAM.

You can create some mapping like this and some memory addresses are reserved for I/O registers. What is the advantage of doing this memory mapped I/O? Then you can simply use your load store instructions to access these registers just like you access memory locations, it just makes it easier. So, no matter which method you use. In the end, the CPU will read and write these registers in order to communicate with I/O devices.

(Refer Slide Time: 23:30)



So, let us just see some examples of this communication? How does this communication look like? So, for example, the CPU wants to read a block of data from a hard disk, how does it happen? So, you have the command register, and then you have the data register and then you have the status register in your device controller. Then the CPU will first write a command into the command register saying read say block number 50. Read this block for me.

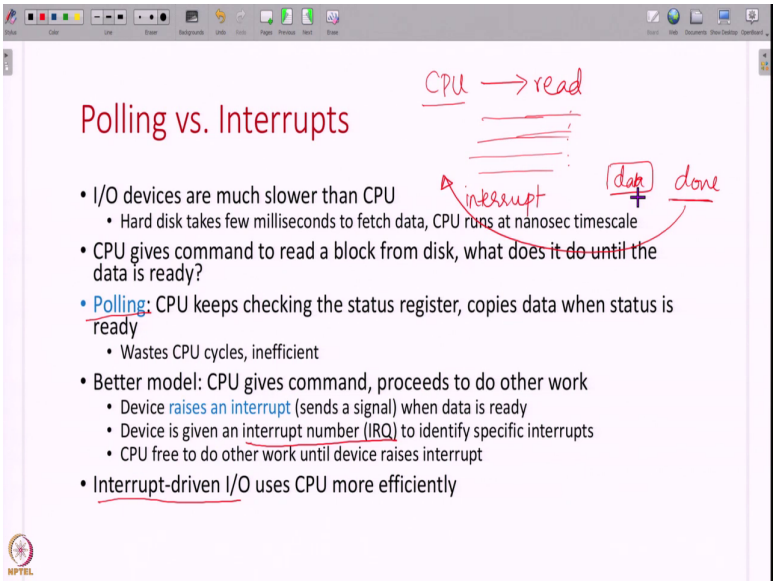
It will give this command and after some time the disk takes a long time, after some time the data is available in the data register. And the device controller will set a status saying done. In the status register it will write some special value saying I am done. So, now the CPU can read this data, once the status indicates it is done.

This is an example of how simple reading happens via accessing these registers in the device controller. Similarly, when you want to write a block, it is similar, it will the CPU will issue a command saying right. So, when you have to write a block the CPU will issue a command saying write and it will also provide the

data immediately, the command the data will be provided immediately because this is the data you have to write.

And then after some time, the device controller will say once the data has been written, the device controller will set a status saying I am done. So, this is how writing a block to hard disk happens.

(Refer Slide Time: 25:09)



So, now let us understand another concept called polling or interrupts. So, I/O devices are much more slower than the CPU. So, the hardest might take few milliseconds to fetch data to read or write data whereas the CPU runs at nanosecond timescales. So, once the CPU has given a certain command saying read, it will take a lot of time before the data is actually ready and the status indicates done.

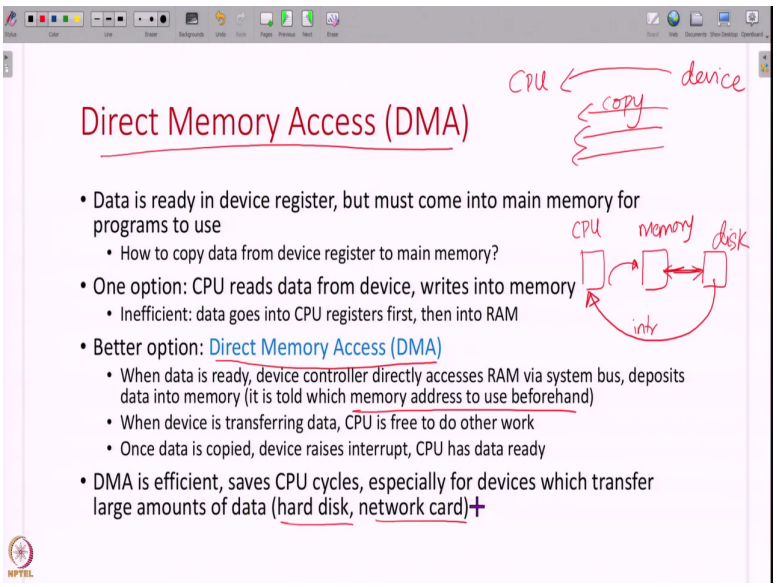
So, in this time, what should the CPU do? There are two options the CPU can keep polling, it can constantly keep checking, is it done, is it done, is it done, is it done constantly reading these registers seeing is it done or a better model could be, I mean you can see this polling is clearly not ideal, because the CPU is wasting a lot of time. It is wasting many milliseconds, it is wasting millions of

cycles. A better model could be the CPU can just issue this command and go away, run some other process or do something else.

And when the device is done, it will then raise an interrupt with the CPU, the CPU is doing something else, then the device once it is done, it will raise an interrupt then the CPU will stop whatever it is doing and it will copy the data from the device then. So, this is called interrupt driven I/O and this is the most commonly used way of communicating with I/O devices. It uses the CPU more efficiently.

So, every device is given an interrupt number called the IRQ, interrupt request number and when its command is executed, its job is done. It will raise an interrupt with its number, say if it is a disk it has given some interrupt number 10, it will say hey, interrupt signal 10 is being sent to the CPU and the CPU can do other work till the interrupt comes and when the interrupt comes it can go and access the data. So, this is the concept of interrupts.

(Refer Slide Time: 27:11)



So, the next concept I want to introduce when it comes to I/O devices is the concept of Direct Memory Access or DMA. So, what is Direct Memory

Access? Now, we have seen in the previous slide, the CPU gives a command to read data when the data is ready, the device will raise an interrupt. So, once the device raises an interrupt to the CPU, the data is available in the data register. Then now, how does the CPU access this data?

Now, the CPU can say copy, copy, if there are you know 512 bytes then copy all this data one by one, the CPU can start accessing the data. But this is wasteful, why is this wasteful? You have your I/O device, you have CPU and eventually all data must be stored in memory, in the memory of the process. So, the device has the data, it will first be copied into CPU registers, caches whatever and then the CPU will copy it into memory.

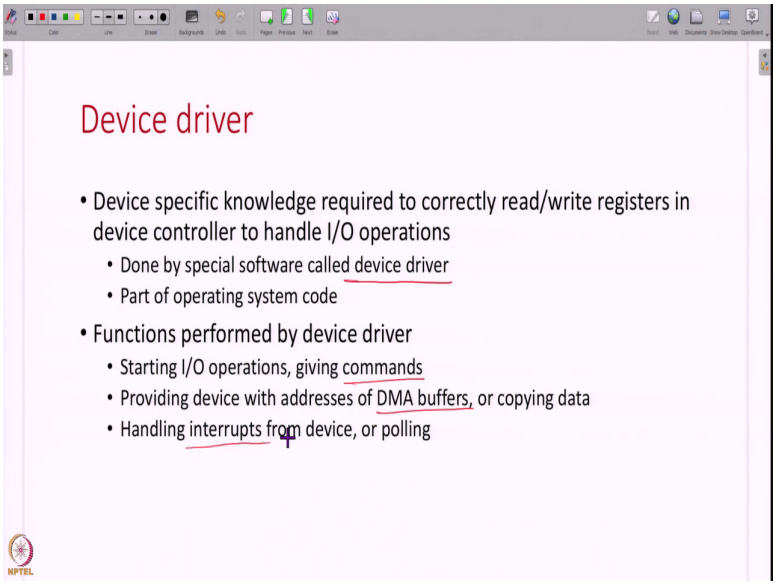
So, this is two hops and it is somewhat slow, you are once again wasting CPU time to do all of this. So, a better idea is what is called Direct Memory Access or DMA. That is the disk or the I/O device directly accesses memory over the memory bus and deposits the data directly into memory location and then it will raise an interrupt to the CPU.

And then once the interrupt is raised, the CPU does not have to do much it has to just see or write the data is already there in RAM and it can just handle the data immediately right. So, how does the disk know where to deposit the data into RAM when the CPU gives the command it can tell the device, it can tell the device which memory address you have to use.

So, there is the small location in memory that I have identified for you. When you finish this command, please deposit the data there. And why in this transfer is happening? The CPU is not involved, the CPU is free to run other processes CPU cycles are not wasted. When the device is transferring the data.

And after the data is copied when the interrupt is raised, the interrupt handling itself is very fast, the CPU does not have to do much at this point. So, DMA is an efficient way of transferring data from I/O devices directly into memory. And it saves CPU cycles because your CPU can actually do other things while this data copy is happening, and the data directly goes into memory instead of first coming to CPU and then jumping into memory. And this is especially efficient for devices like hard disk and network card that transfer a lot of data at a time. So, this is the concept of direct memory access.

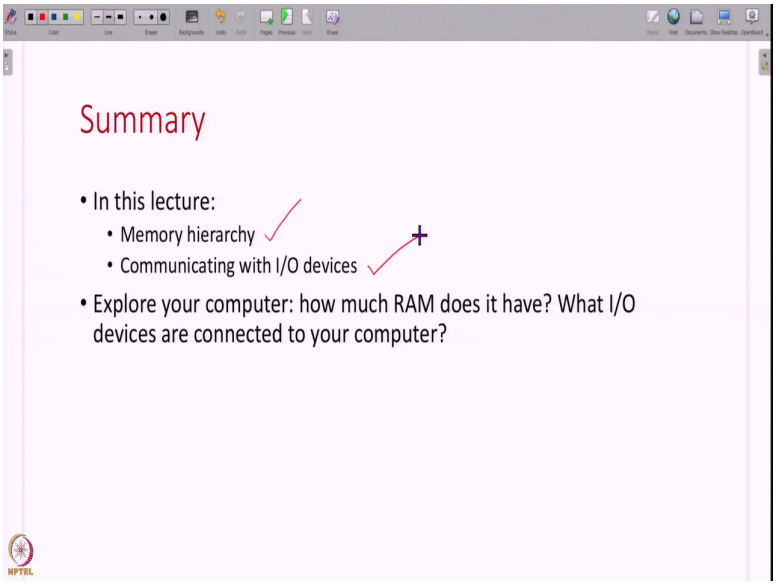
(Refer Slide Time: 29:52)



And all of these communication with an I/O device, reading and writing registers, handling interrupts, giving these commands, identifying memory locations for DMA. All of this is done by a special piece of software that is part of the operating system which is called the device driver.

And so that you as a user program, you do not have to worry about how this is happening. And some of the functions performed by the device driver are giving commands, figuring out which registers to read or write, setting up these DMA buffers, handling interrupts all of this is done by the device driver.

(Refer Slide Time: 30:32)



So, in summary, in this lecture, we have studied the memory hierarchy, the storage hierarchy in a computer system starting from registers to DRAM to CPU caches, hard disk everything. And we have also studied various concepts around how communication with I/O devices happens.

So, as a practical exercise, go back, if you have a computer that you can access, look at how much RAM does it have, what all I/O devices are connected, can you figure out what IRQ lines, what interrupt numbers are assigned to them? So, using a few commands in Linux, you should be able to get all of this information and this is a small practical exercise for you to understand the concepts of this lecture better. So, thank you all. With this, I would like to finish this lecture. We will continue our course in the next lecture. Thank you.