Design and Engineering of Computer Systems

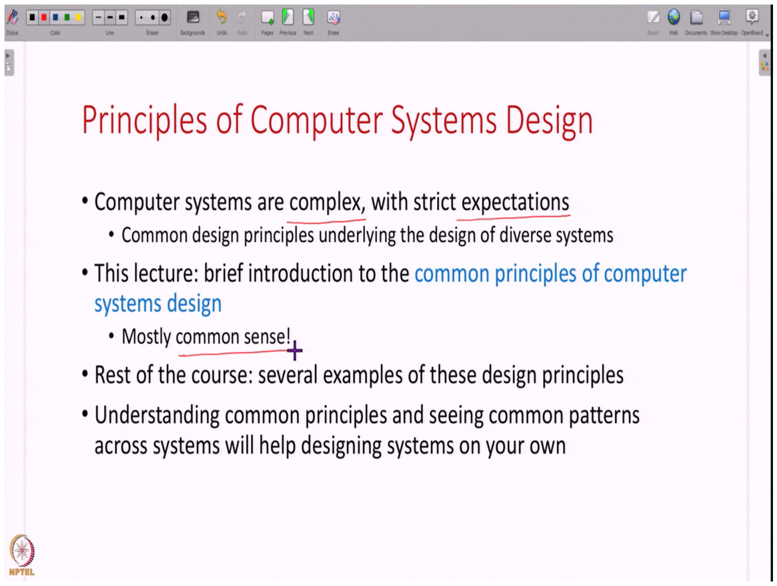
Professor Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology, Bombay

Lecture 2

Principles of Computer System Design

Hello everyone, welcome to the second lecture in the course Design and Engineering of Computer Systems. In this lecture what we are going to study are some of the principles underlying the design of Computer Systems.

(Refer Slide Time: 00:30)



Principles of Computer Systems Design

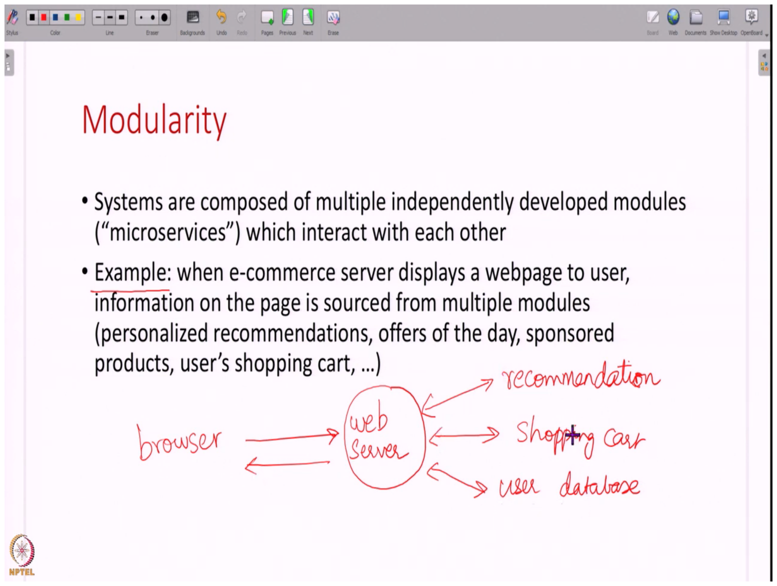
- Computer systems are complex, with strict expectations
  - Common design principles underlying the design of diverse systems
- This lecture: brief introduction to the common principles of computer systems design
  - Mostly common sense!
- Rest of the course: several examples of these design principles
- Understanding common principles and seeing common patterns across systems will help designing systems on your own

So, let us begin, as we have seen in the previous lecture Computer Systems are fairly complex and they come with certain expectations in the form of functional and non-functional requirements. So, real life systems can look very diverse but underlying all of them are of a common set of design principles.

So, as we will study in the course we will study different types of systems how to build them and all of that but before we begin any of that in this lecture what I would like to present to you are some common design principles. So, no matter which system you look at these are some common-sense principles that are important for us to know. And once we see these principles you will keep seeing them again and again and then you will think ok this is the pattern this is the pattern you will be able to recognize patterns.

And this will help you to be able to design your own systems in the future. So, from the next lecture onwards we will get into specifics starting with hardware, operating system, application design, performance, reliability and everything. But this lecture we are going to study some of the common principles that we will see throughout the rest of this course. And of course, there is nothing very sacred about these principles these are all just common sense, but seeing them in a concrete form now will help you see the pattern again and again in the future.

(Refer Slide Time: 02:02)



Modularity

- Systems are composed of multiple independently developed modules ("microservices") which interact with each other
- Example: when e-commerce server displays a webpage to user, information on the page is sourced from multiple modules (personalized recommendations, offers of the day, sponsored products, user's shopping cart, ...)

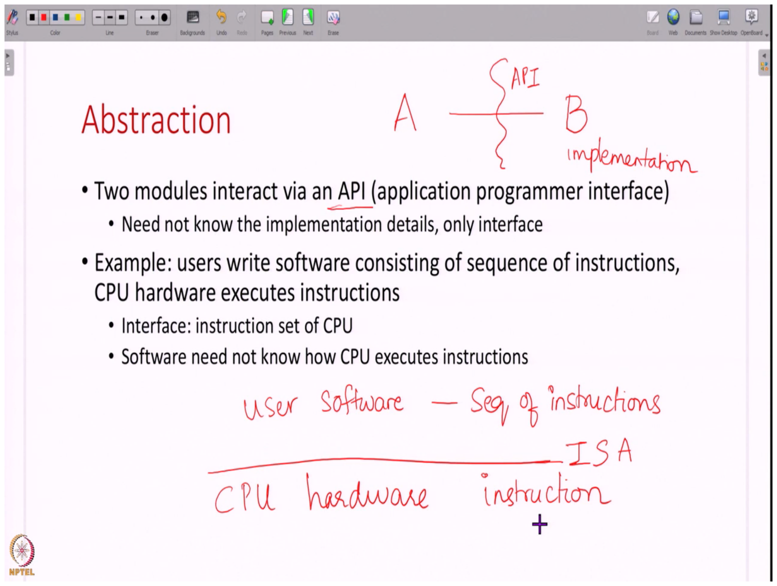
So, let us start with our first principle, our first principle is Modularity. So, what is Modularity? Real life systems we have seen are big complex so instead of building them all in one go you will build them as smaller modules and put them together these modules will then interact together. So, this idea is also called microservices or you know when you write your C program you use different functions to write your code all of these are examples of modularity in Computer Systems.

So, one concrete example we have seen is an e-commerce server in the previous lecture, so in the case of an e-commerce server how does modularity work it is as follows. You will have suppose you have the user's browser has requested a web page from an e-commerce server. So, there is a web server a machine that listens to your request and gives you a response.

Now, this web server need not provide all the functionality of an e-commerce website in this machine itself, for example, there could be another server that is generating recommendations this user has purchased these items in the past and this time he is likely to purchase these items. So, there is somebody else doing the recommendation so this web server will get the recommendations from one server or there could be another server that is serving the offers of the day another server that is doing sponsored products and you know maybe another server is maintaining your shopping cart and another server has all your payment and billing information there is a user database.

So, there could be many different servers each providing separate functionality and the web server is talking to all of them assembling the response to give to you and then returning a response to. So, real life systems are typically decomposed like this into multiple microservices or modules and they all interact with each other to deliver the service to you.

(Refer Slide Time: 04:11)



Abstraction

- Two modules interact via an API (application programmer interface)
  - Need not know the implementation details, only interface
- Example: users write software consisting of sequence of instructions, CPU hardware executes instructions
  - Interface: instruction set of CPU
  - Software need not know how CPU executes instructions

So, the next principle which is related to modularity is what is called Abstraction. Now, when you do Modularity when you follow Modularity you split your system into multiple modules how should these modules interact with each other? So suppose you have module A and that is interacting with module B.

So, if module A wants to talk to module B there is a certain interface or Application Programmer Interface or API between these modules. Now, if module A can use the services of module B only by knowing this API or interface without needing to know any details about the implementation of B. A does not know how B is implemented it just knows what is the functionality what is the interface being exposed by B.

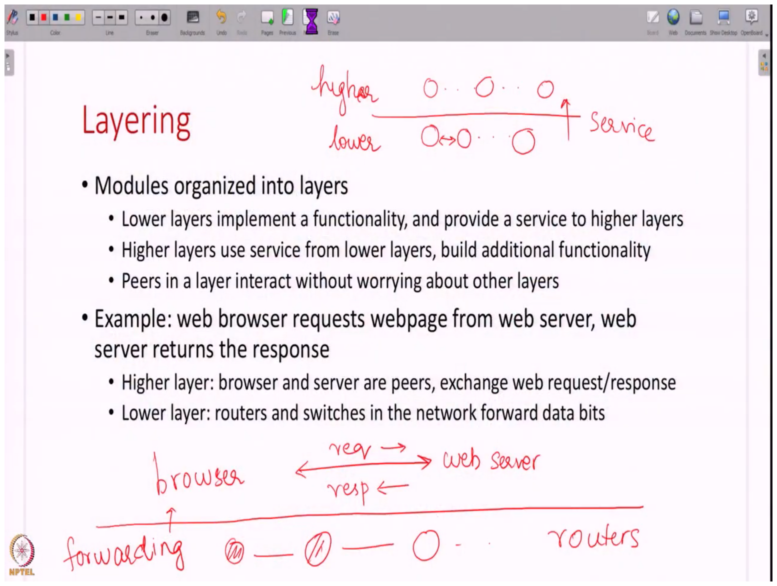
By just knowing the interface the API if you can use a module then that design is said to have a good Abstraction. So, this is the concept of an abstraction you are abstracting out the details of module B and exposing only an interface. So, there is a concrete example for this that is working for you in everyday life which is you know whenever you write software you write your C program it is eventually compiled into a sequence of instructions which run on the CPU. So, you have computer hardware you have your CPU hardware, and this CPU hardware executes certain instructions.

These instructions could be get this piece of data add these two numbers subtract these two numbers whatever there are many different instructions that each CPU executes and you as a user when you write your software, you write your software as a sequence of instructions, and then the CPU will execute these instructions.

So, a user does not need to know how the CPU implements these instructions. You do not know how your underlying CPU hardware is actually adding two numbers what are all the various electronic circuitry involved over there. That is not needed. You can simply write code which has an instruction to add two numbers as long as you know this is called the Instruction Set Architecture or ISA.

As long as you know this interface this API that your hardware is providing you as a user can write software that runs on this hardware it is not necessary for you to know the implementation of how these instructions are implemented in the underlying hardware. So, this is a very simple example of the concept of abstraction and this has to be used widely otherwise there is no way you can build these complex Computer Systems without a good amount of abstraction.

(Refer Slide Time: 07:04)



Layering

- Modules organized into layers
  - Lower layers implement a functionality, and provide a service to higher layers
  - Higher layers use service from lower layers, build additional functionality
  - Peers in a layer interact without worrying about other layers
- Example: web browser requests webpage from web server, web server returns the response
  - Higher layer: browser and server are peers, exchange web request/response
  - Lower layer: routers and switches in the network forward data bits

So, the next principle is Layering. So, what is Layering? It is also related to modularity so when you split your system into multiple modules you will organize them into layers. So, the modules in the lower layer will implement some functionality and they will provide a service to the modules in the higher layer.

And the modules in the higher layer will use this service without worrying about how the service is implemented. And they will add more functionality which they will provide to the modules in the further higher layers and so on. So, peers within a layer interact with each other but across layers, you just interact via the service. So, again, this is a different way of enforcing modularity. So, again a common example is the internet the internet is built as layers. So, there are higher layers and there are lower layers. So, what are these layers in the internet let us see an example.

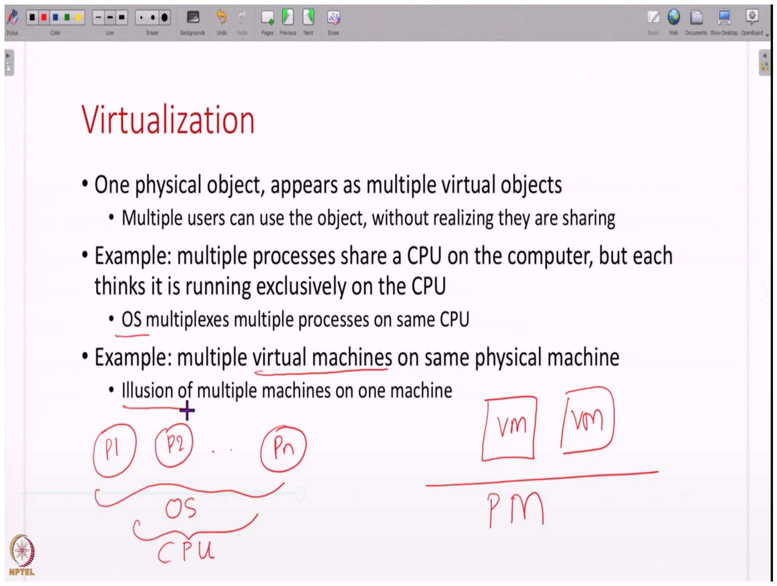
So, suppose you have a web browser that is accessing a website downloading a web page from a web server and you know the web browser sends a request and the web browser sends a response back to you that you view on your browser. Now, this browser and the server are peers they are just exchanging you know, hey give me this web page request response and so on.

But all of this information that is there in the request or responses basically some messages some series of bits underneath this layer of the browser and the web server there are routers in the internet. There are a series of routers in the internet routers and switches which are actually forwarding these bits of information from your browser from your laptop or desktop to a web server somewhere on the internet.

So, what are these routers doing? They are doing the forwarding. So, this is a lower layer they are providing a service of forwarding to the higher layers. So, the browser does not really know how a router is doing its job a browser just sends a request gets a response displays the web page a router really does not know how a browser or web server works its job is to forward traffic forward bits and it is doing that job.

So, this is how you design a complex system in layers you design the routing layer independently you design the browser and web server layer independently so that when you put them all together they work well. So, this is the concept of layering you will see several examples of this principle in fact all of these principles you will keep seeing examples throughout this course.

(Refer Slide Time: 09:58)



Virtualization

- One physical object, appears as multiple virtual objects
  - Multiple users can use the object, without realizing they are sharing
- Example: multiple processes share a CPU on the computer, but each thinks it is running exclusively on the CPU
  - OS multiplexes multiple processes on same CPU
- Example: multiple virtual machines on same physical machine
  - Illusion of multiple machines on one machine

So, the next principle is what is called Virtualization. So, what is virtualization? In common English terms, it means that there is one physical object but you somehow make it appear as multiple virtual objects you give an illusion of multiple virtual objects. So that multiple users will keep using these virtual objects without realizing that they are all sharing the same underlying physical object that is the principle of virtualization.

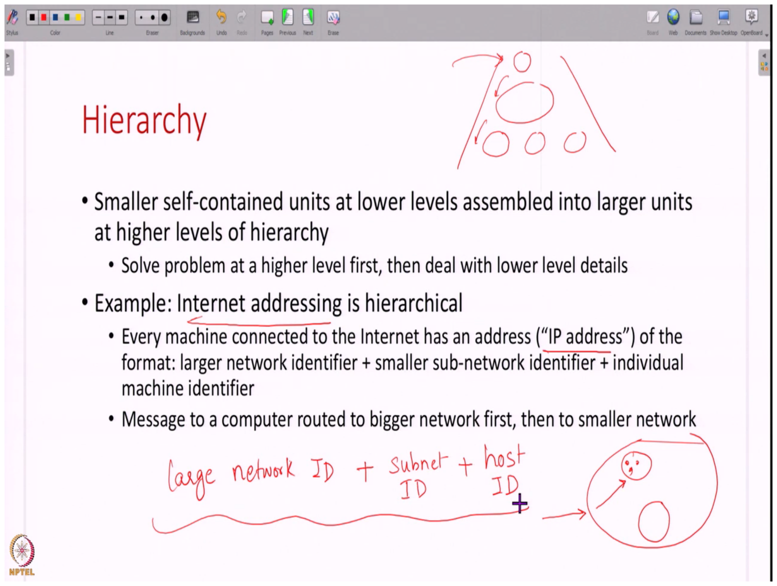
So, what is an example of this is you know on your computer there are typically multiple programs running programs are also called processes we will understand what the word process means later but there are multiple programs running on a CPU. And each program is just thinking that it is running exclusively on the CPU you might have a computer game running you might have a web browser running a movie streaming all of these are running on the CPU.

But each of them is not aware that the other processes are there each process thinks I have full control over the CPU and I am running on the CPU. But the underlying operating system basically takes care of if you know if you have multiple programs P1 P2 and so on Pn the operating system somehow multiplexes all of these on the underlying CPU. So that the processes are not aware that they are only getting a slice of the CPU. So, this is the way we say this is the operating system virtualizes the CPU and gives a slice to each of these different programs that are sharing the CPU.

So, we will see how this is done there are mechanisms to do this but this is a common design principle that makes it easy for us to share Computer Systems across multiple users. Another example is that of virtual machines if you have ever used a virtual machine you know you run say Linux inside a virtual machine on a windows laptop then you know what virtual machines are. With a virtual machine also, the concept is similar.

So, you can set up multiple virtual machines on an underlying physical machine there is one computer system one laptop with one CPU memory everything but the multiple virtual machines think that they have the full machine for themselves full underlying machine for themselves. So, you give an illusion so how we do this we are going to study later in the course but this principle of virtualization itself is important to be able to share Computer Systems efficiently.

(Refer Slide Time: 12:32)



Hierarchy

- Smaller self-contained units at lower levels assembled into larger units at higher levels of hierarchy
  - Solve problem at a higher level first, then deal with lower level details
- Example: Internet addressing is hierarchical
  - Every machine connected to the Internet has an address ("IP address") of the format: larger network identifier + smaller sub-network identifier + individual machine identifier
  - Message to a computer routed to bigger network first, then to smaller network

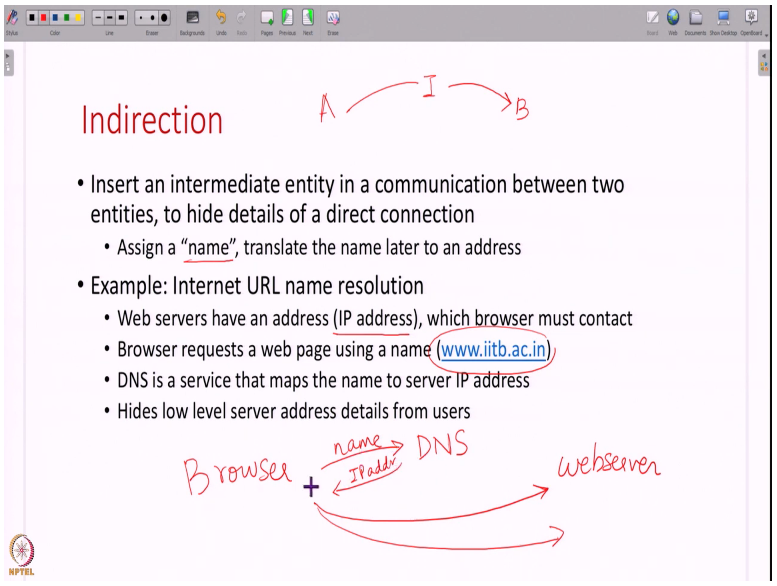
So, the next design principle is that of Hierarchy. So, intuitively we all understand what hierarchy is you know there are certain smaller units that assemble to form larger units. So, there is a smaller set of units then there will be a bigger unit then be a bigger unit you typically have a hierarchy. And whenever you have to solve a problem for example in your institute there are students then there are you know student leaders and one head of all the student organization so whenever there is any problem you first go to the higher level then you solve it there and come down. So, hierarchy gives you an easy way to manage complexity in a large system.

So, one example from Computer Systems is Internet Addressing. So, every machine that is connected to the internet your laptop desktop server everything has an address called the Internet Protocol or IP address. So, this IP address is a hierarchical address. If you look at IP address it will have multiple components it will have the larger network identifier, there will be a big network and you will put that network identifier. Then you will concatenate it with the smaller subnet you know smaller network identifier then the actual host machine itself the machines identifier together you will create an IP address for a machine. So it is hierarchical.

So, whenever somebody has to send a message to your computer they will first look at this large network id, send the message to the bigger network, then that bigger network will identify a smaller network you know, in this way, if the first message comes to the bigger network it will identify the smaller part inside the big network, there are smaller networks, inside that there are hosts. So message first comes to the bigger network and then it goes to the smaller network and then it goes to the individual host.

So, this will ensure that routing everybody does not have to know about everybody else you just know let me just send this message to this bigger network and it will find its way there. So, this makes routing traffic on the internet easy we are going to study this in more detail but this principle of the hierarchy itself is very powerful and is used in many places across Computer Systems.

(Refer Slide Time: 14:53)



Indirection

- Insert an intermediate entity in a communication between two entities, to hide details of a direct connection
  - Assign a "name", translate the name later to an address
- Example: Internet URL name resolution
  - Web servers have an address (IP address), which browser must contact
  - Browser requests a web page using a name (www.iitb.ac.in)
  - DNS is a service that maps the name to server IP address
  - Hides low level server address details from users

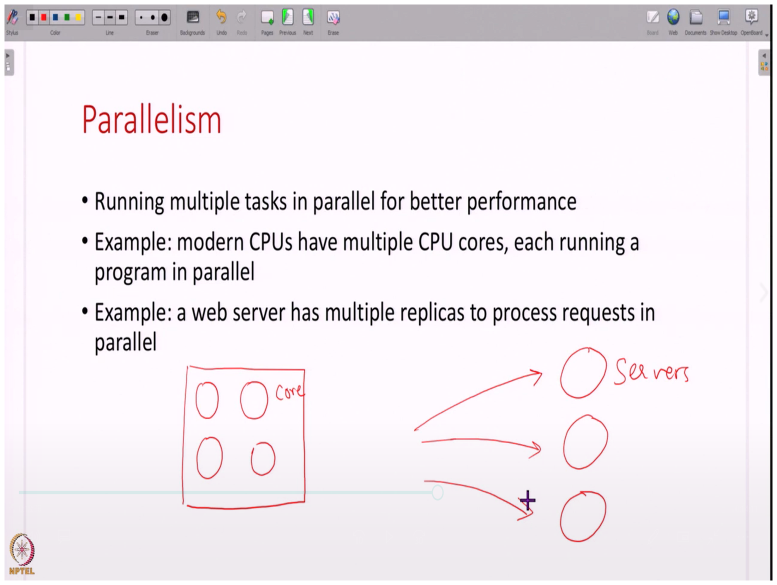
So, moving on the next principle is what is called Indirection. Indirection simply means you know you add an intermediate entity and indirectly talk to somebody else. If you know A wants to talk to B, instead of talking to B directly, it will talk to some intermediate I which will then talk to B instead of a direct communication. So, why is this there are many benefits of doing this of course this adds to the overhead but there are many benefits of doing Indirection. For example, one popular example is you know you assign a name and then resolve the name to an address.

So, again let us come back to the internet example we have seen that every machine has an IP address but you know this is a large number and remembering this number is a little cumbersome. And if you want to for example access some content from NPTEL you have to talk to NPTEL's web server you have to know the address of that web server which is difficult to remember.

So, what people remember is they remember the name, the name of the web server for example IIT Bombay's web server might have a name like www.iitb.ac.in and your browser in your browser's top part of the browser you will type this name and then this browser will contact your browser will contact something called DNS which resolves this name and it will give you an IP address of the server. And then your browser can go and contact the web server, can send a message to the web server using hierarchical routing the message will reach the web server.

So, this DNS is adding one level of indirection. You give a name it will give you an address and then you can talk to the address. So, by adding this intermediate layer you are simplifying things you know users do not have to remember this cumbersome address and if you know IIT Bombay wants to move to a different website it can simply update this mapping in DNS you as a user you do not have to be aware that the web server has changed you will simply get a different IP address you will go to the different type address. So, indirection makes things easy. So this is again a common principle you will see throughout Computer Systems in many places.

(Refer Slide Time: 17:13)



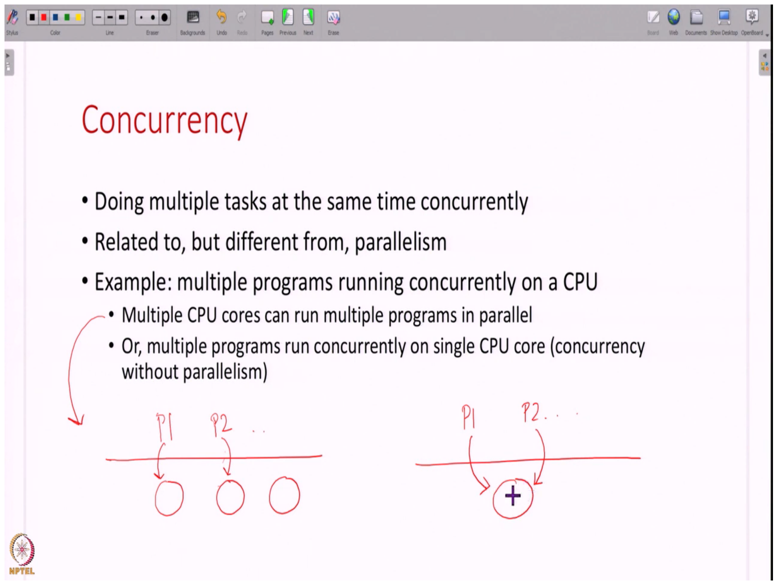
Parallelism

- Running multiple tasks in parallel for better performance
- Example: modern CPUs have multiple CPU cores, each running a program in parallel
- Example: a web server has multiple replicas to process requests in parallel

So, moving on the next design principle is that of Parallelism. What is parallelism again intuitively we all understand to do things in parallel and you will get better performance. So, modern CPUs today, if you buy a CPU today, modern CPUs have multiple CPU cores. Even your phone has multiple CPU cores and laptops and desktops have you know four, eight even more CPU cores, servers have tens to hundreds of CPU cores in one CPU box. And each of these CPU cores can run a program in parallel you know one CPU core is running is streaming movies the other is a web browser the other is an email client they can do multiple tasks in parallel to get better performance.

So, this is one common example of parallelism another example is you know you can even have parallelism at the level of servers for example if you go to a large website the website has multiple web servers okay multiple replicas of web servers. And whenever a web request comes some web requests some requests are handled by the server some by the server some by the server, so that in parallel you can serve a large amount of traffic so that no one server gets overloaded. So, this is the concept of parallelism to do things in parallel usually for better performance.

(Refer Slide Time: 18:40)



Concurrency

- Doing multiple tasks at the same time concurrently
- Related to, but different from, parallelism
- Example: multiple programs running concurrently on a CPU
  - Multiple CPU cores can run multiple programs in parallel
  - Or, multiple programs run concurrently on single CPU core (concurrency without parallelism)

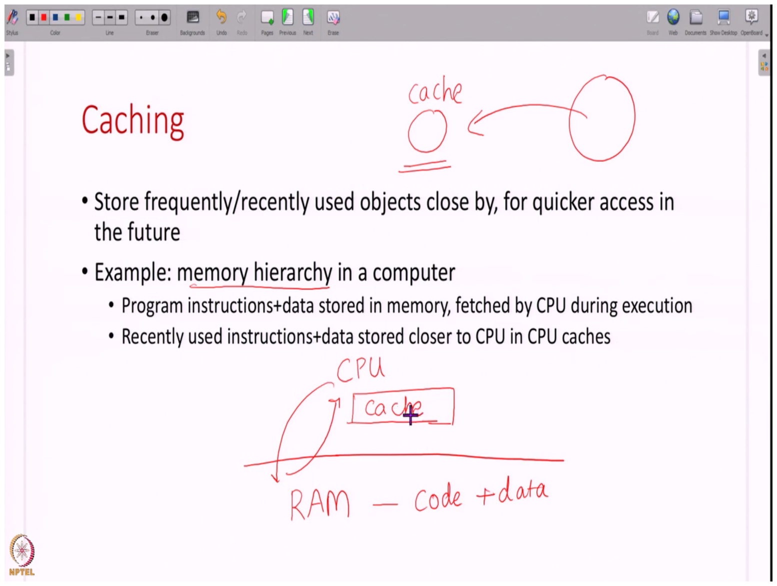
So, the next principle is that of Concurrency. Concurrency means to do anything concurrently means to do multiple things at the same time that is called I am working on five different things concurrently I am doing them at the same time. So, if you are wondering but that is parallelism you have just spoken about it no there is a subtle difference between they are related but there is a subtle difference between Concurrency and Parallelism.

So, for example, if I am running multiple programs on my machine and on my computer, I am running multiple programs and if I have multiple CPU cores and if I have process P1 running on one core P2 running on one core they are all running in parallel that is parallelism. On the other hand, if I only have a single core or on one core I will run processes P1 P2 multiple processes on one core in a time sharing fashion you know that is called concurrency.

So, this is concurrency without parallelism. So I can concurrently run programs in parallel like this or I can concurrently run programs even when there is no parallelism by quickly switching between them. So, modern CPUs do this all the time they run one process for some time stop it run something else, stop it run something else stop it so that all of these processes think they are running on the CPU but in fact they are sharing the CPU this is related to the concept of the operating system virtualizing the CPU for processes we have studied that.

So, you have multiple processes running concurrently even without parallelism, this is a very important concept that we will come back to again and again in the course so whenever you want to do things efficiently you have to do multiple things at the same time concurrently for good performance.

(Refer Slide Time: 20:33)



Caching

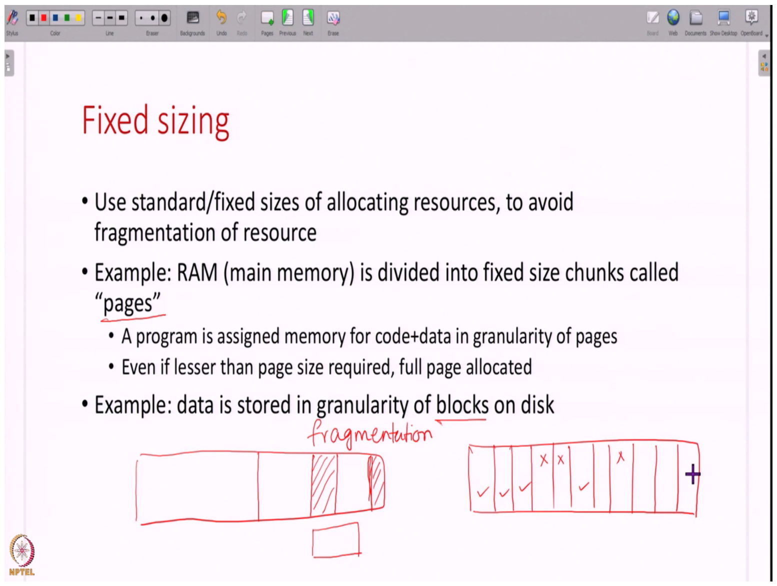
- Store frequently/recently used objects close by, for quicker access in the future
- Example: memory hierarchy in a computer
  - Program instructions+data stored in memory, fetched by CPU during execution
  - Recently used instructions+data stored closer to CPU in CPU caches

So, moving on the next design principle is Caching. So, what is Caching? So, if there is some information that is very far away i.e. some object far away and you have gotten it over a long distance and you used it once then you might want to keep this. You know there is some object you have gotten it you have gotten a small slice of it closer, you might want to save this copy in a cache so that in the future you do not have to get it again for quicker access in the future.

So, a common example for that is the memory hierarchy in a computer. So, in a computer you have your RAM or your main memory that has all the code instructions in a program plus all the data in a program is stored in your RAM. And then your CPU gets these instructions gets this piece of data and executes them, you have written a program with some code the CPU runs the code. But every time the CPU has to go to the RAM fetch an instruction fetch a piece of data and this distance between the CPU and the RAM it takes a lot of time. So what modern CPUs do is they have a cache.

So, in the cache the recently used instructions and data that have been fetched from ram they are stored in the cache so that in the near future if the CPU needs the same instruction or the same data again it does not have to go all the way to memory it can simply use it from the cache. So, this is the concept of caching and not just in CPUs many systems you know databases have caches in front of them so that if you have gotten a piece of data from a database after a lot of searches then you just store that data in a cache so that you can quickly access it in the future. So, this is a common idea that will come back to again and again in the course.

(Refer Slide Time: 22:23)



Fixed sizing

- Use standard/fixed sizes of allocating resources, to avoid fragmentation of resource
- Example: RAM (main memory) is divided into fixed size chunks called "pages"
  - A program is assigned memory for code+data in granularity of pages
  - Even if lesser than page size required, full page allocated
- Example: data is stored in granularity of blocks on disk

So, the other interesting design principle that Computer Systems follow is what is called Fixed Sizing. So, suppose you have a resource and you are distributing the resource amongst multiple users instead of giving variable sized quantities to each user it might be more efficient if you just break up the resource into fixed sizes and give it to users this avoids the fragmentation of resources.

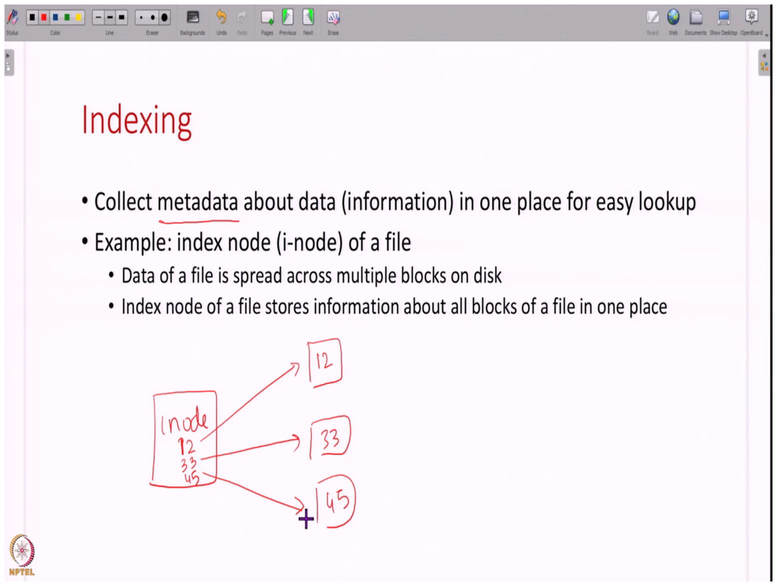
For example, if you have some memory somebody has taken this chunk of memory somebody has taken another chunk of memory, a smaller chunk of memory and somebody has taken so much memory and you are left with this small piece over here. And then somebody has freed up this chunk in between so now you have this memory is in use, this is in use, this part is free and this part is free.

Now, if somebody requests a memory of this size you cannot give it from here or here it does not fit anywhere so this is called in general Fragmentation. Your resource gets fragmented with time as you use it free it up you will be left with these small holes and all sorts of disorganized fragments this is very inefficient.

So, instead, a better way would be today in Computer Systems your main memory or RAM is divided into fixed size chunks. There is a standard size in which memory is given out that is called pages. There is a fixed size chunks that memory is divided into and if a program wants some memory it is given some number of chunks. If a program wants some memory it is given these chunks, some other program is given these chunks you will only get some integer number of pages. Even if you are using lesser than a page you will get an entire page for you to use. Because it just makes memory allocation easier so now somebody else comes you give some of the free chunks.

So, memory is allocated and freed up at this fixed size granularity called pages. Similarly, on the disk also data is stored in fixed size blocks. All of this makes it easy to allocate and free up memory and fix size and also non-contiguous. It is not like everything is contiguous. Wherever there is a page available, wherever there is a block available you will give it to the program or you will give it to a file. This makes it easy to manage resources, it makes it efficient, it also avoids fragmentation.

(Refer Slide Time: 24:51)



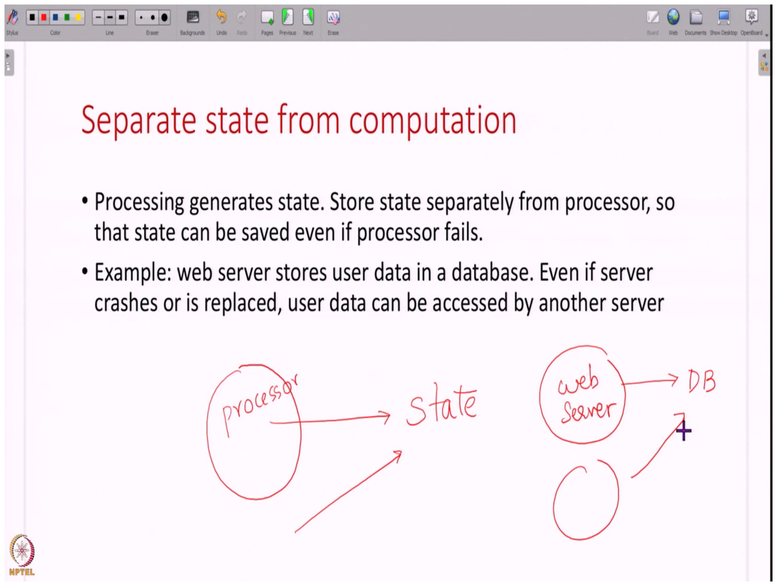
Indexing

- Collect metadata about data (information) in one place for easy lookup
- Example: index node (i-node) of a file
  - Data of a file is spread across multiple blocks on disk
  - Index node of a file stores information about all blocks of a file in one place

So, the next principle is that of Indexing. Now, if the information about an object is spread across multiple locations then you have to maintain it. It is easy for you to maintain all of this information. So the information about a piece of data is called metadata. So, you collect all this metadata and store it in one location which is called the index node or the i-node of a file. For example, in a file system, your file is split into multiple locations. We have seen that the disk space is allocated in the granularity of blocks say suppose block numbers 12, 33, 45 and you know different blocks the information of your file is spread out.

So, then what you do is you maintain one index node or i-node of a file which has all this information 12, 33, 45 all of this information about a file you know pointers to all of these blocks are stored in this i-node such that it is easy for you to look up this file. Whenever in the future you want to access this file you do not have to look all around the disk to see where the file is you just go to the i-node you have all the information about the file in the i-node. So, this concept of indexing is used in databases it is used in many places throughout computer systems where you collect all the information about data, all the metadata about data, in one place for easy access.

(Refer Slide Time: 26:20)



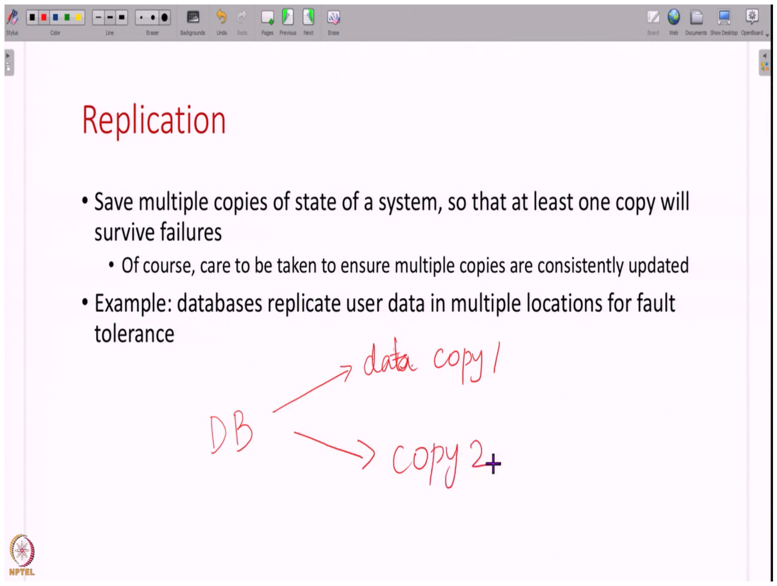
Separate state from computation

- Processing generates state. Store state separately from processor, so that state can be saved even if processor fails.
- Example: web server stores user data in a database. Even if server crashes or is replaced, user data can be accessed by another server

So, another common design principle is that of Separating state from computation. So, there is a program that is running it has done some calculations and it has generated some state. State is nothing but the data variables whatever information has been generated. So, the CPU or the processor is generating state and you would like to maintain this processing and the state separately. So that even if the processor fails then somebody else can come and access the state if a server fails then another server can come and access the state as long as the state is safely stored somewhere.

So, with this is what you know web server stores you know if you go to an e-commerce website all your account information is stored in a database somewhere so that even if this server crashes another web server can go and get your information from the database. So, separating state from computation is also a common design principle we will see in multiple systems simply from the point of view of fault tolerance it makes a lot of sense.

(Refer Slide Time: 27:25)



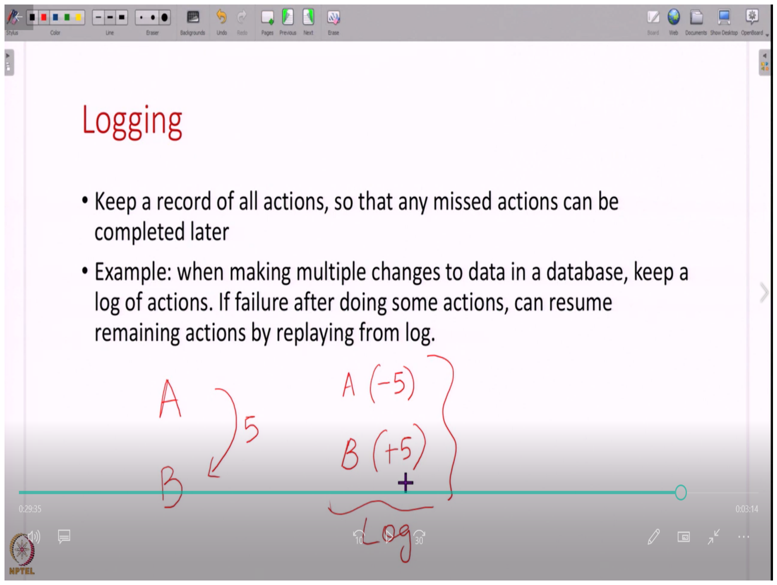
Replication

- Save multiple copies of state of a system, so that at least one copy will survive failures
  - Of course, care to be taken to ensure multiple copies are consistently updated
- Example: databases replicate user data in multiple locations for fault tolerance

$$\begin{tikzcd} & \text{data copy 1} \\ \text{DB} \arrow[ur] \arrow[dr] & \\ & \text{copy 2} \end{tikzcd}$$

Then the next design principle is that of Replication. So, what is their application? When you separate, when you do the separation of state from the computation, instead of storing one copy of the state for example a database it does not just store one copy of the data. It will store multiple copies of the data so that even if one copy is lost due to say a hard disk failure the other copy remains. So, replication is also widely used for fault tolerance in Computer Systems.

(Refer Slide Time: 28:00)



Logging

- Keep a record of all actions, so that any missed actions can be completed later
- Example: when making multiple changes to data in a database, keep a log of actions. If failure after doing some actions, can resume remaining actions by replaying from log.

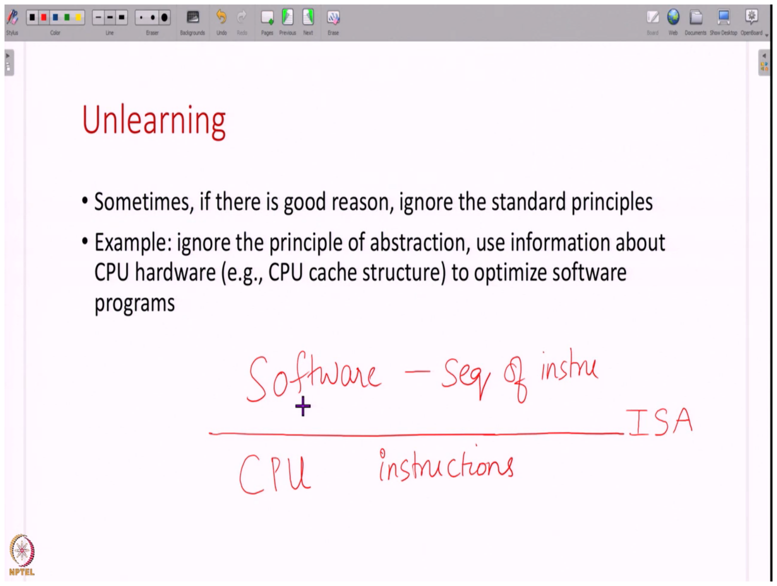
$$\begin{tikzcd} A \arrow[d, bend left, "5"] \\ B \end{tikzcd} \qquad \underbrace{\begin{matrix} A\ (-5) \\ B\ (+5) \end{matrix}}_{\text{Log}}$$

Another design principle is that of Logging. So, Logging simply means you keep a record of all your actions so that even if a failure happens some action is missed you can always redo it later so this is a very simple but powerful idea. So, suppose you know you are a banking application and you are transferring some amount of money from account A to account B, so you are transferring say 5 units of money from account A to account B. So what you will do is when you are making this transfer you will first write in a log from account A I am deducting 5 units from into account B I am adding 5 units. You will first note this down and then you will go and make the changes to the account.

Why is that so suppose you did not do this you deducted money from account A and then a power failure happened and you forgot about it that you never added back money into account B. That is wrong and then the people will complain. But if you store this in a log and then you start making these changes what happens suppose you deducted money from account A and you crash then after the computer is restarted after a crash you can see ok there is more to be done that I did not do and let me do that.

So, logging is a very powerful idea in order to ensure you know remember this concept of atomicity we talked about you make a payment to an e-commerce firm it has to ship the item to you it cannot just take the payment crash and then forget shipping. Therefore, if you have to do multiple actions together and you want all of them to be completed atomically then logging is a very powerful idea.

(Refer Slide Time: 29:45)



Unlearning

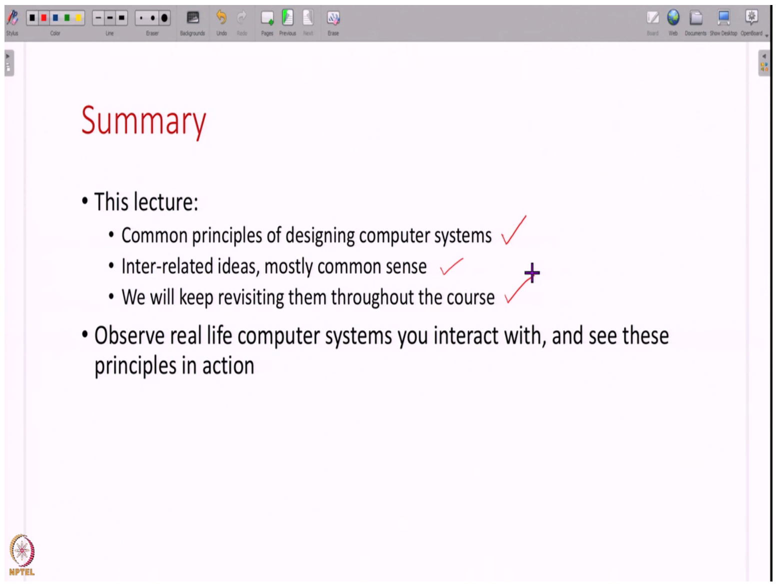
- Sometimes, if there is good reason, ignore the standard principles
- Example: ignore the principle of abstraction, use information about CPU hardware (e.g., CPU cache structure) to optimize software programs

So, finally, the last design principle I am going to leave you with is sometimes there is a good reason if there is a good reason feel free to ignore these standard design principles violate them and you know do something the opposite of what the design principle says. And this is again commonly done in Computer Systems ironically. For example, at the beginning of this lecture I told you the principle of abstraction. So, there is a CPU hardware this CPU executes instructions and then software is simply written as a sequence of instructions and you do not bother how the CPU works how it is implemented how it executes the instructions you just write a sequence of instructions and give it to the CPU.

And there is an abstraction here there is a instruction set architecture a set of instructions defined by the CPU you only need to know that you do not need to know the internal structure of a CPU that is what we studied in abstraction. But most of the times what happens is if your software has to run properly you have to know the internal details of the CPU. For example, if you know how CPU caches work you can write programs that will perform better that are optimized for the CPU crashes so this is routinely done.

So, this is in some sense violating the abstraction principle but it is needed in order to get good performance, so at any point of time in your system design you feel that some of these design principles are restrictive you can go ahead ignore them and do the opposite of what the principle says. But you should have a good reason all of these principles are there for good reason and therefore for ignoring them you should have a good enough reason.

(Refer Slide Time: 31:26)



Summary

- This lecture:
  - Common principles of designing computer systems ✓
  - Inter-related ideas, mostly common sense ✓
  - We will keep revisiting them throughout the course ✓
- Observe real life computer systems you interact with, and see these principles in action

So, with this I would like to summarize that in this lecture what we have studied is we have studied some common design principles that are useful in designing Computer Systems. So, these are all interrelated ideas there is a lot of overlap and these are all mostly common sense there is nothing rocket science about these principles.

And we will keep revisiting them throughout the course we will keep coming back to these concepts again and again in the course. So, as an exercise for you take some time to observe in real life systems or even not just Computer Systems in any systems you will find these design principles in use. So, try to observe systems try to see these principles in action and understand them fully. So, thank you all with this I would like to conclude this lecture and see you all in the next lecture in this course thank you.