

Operating System Fundamentals
Prof. Santanu Chattopadhyay
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture – 10
Operating System Structures (Contd.)

(Refer Slide Time: 00:29)

System Programs (Contd.)

- **File modification**
 - Text editors to create and modify files
 - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
 - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

The diagram shows a vertical stack of memory blocks. The top block is labeled '10MB' and the bottom block is labeled '1000'. A shaded area in the middle is labeled 'Main'.

File modification system programs. So, for example, text editor is a program by which we can create or modify text files so that is one possibility. Then there are special commands to search contents of files and or perform transformation of the text, so that is another type of file modification that we like to do.

Then programming language support; so, there are compilers, assemblers, debuggers, interpreter. So, they are sometimes provided like some operating system. So, they come up with some built in compilers. So, built in assemblers, debuggers, so like that. So, they are often supported and some operating system they will not have this type of support, but whatever it is, so if it come. So, many cases it comes up with that support.

Then program loading and execution. So, there are different types of loaders that are available like absolute loader, relocatable loader, then linkers, then overlay editor lower loaders. So, these are different type of mechanism by which programs can be loaded into the main memory. So, I will just quickly have a look at that. So, this if you look into a

program. So, it is residing in the disk now from the disk the program for execution has to be brought into the main memory.

So, this is my main memory, now this main memory where exactly we load the program? Suppose I have got a program of size set 10 kilobyte, now that 10 kilobyte may be loaded in this part starting at from the location 1000 onwards or it may be located loaded at some other addresses also. Now, in case of absolute loader what happens is that whenever you load the program there is a fixed address from where it is loaded. So, it is loaded from this address only.

So, this is possible if the number of programs in the system is fixed and we know their load values previously. So, normal like typically for dedicated application like telephone exchange and all. So, where the set of programs that are running is fixed, so for those cases this absolute loaders are useful. Some, but in more generic form of loader is the reloadable loader that is depending upon wherever the memory is free so maybe memory is free in this region. So, instead of loading from 1000 the program will be loaded from this address. So, this way we have got this reloadable loaders.

So, these are their absolute loaders, reloadable loaders. Then there are linkage editor or linkers, so that actually combines a number of different executable modules, so that object modules to get the get one executable module. So, maybe the same parts of applications are developed by different users, they write their different modules and all these modules need to be combined together to get the final executable version. So, that is done by this linkage editor.

Then we have got overlay loaders, where something some part of the program is overlaid by some other part of the program. So, maybe at the beginning of a program, so you are requiring procedure 1, but sometime later we find the procedure 1 is no more required only procedure 2 is required now. So, instead of loading procedure 2 at a different location, so we just use the location that where we had loaded procedure 1, so that is the concept of overlay. Overlay loaders are useful when we have got this space constant.

Now, debugging systems for higher level and machine language so, this is also a support that is provided because we can debugging it is better that if we can do the operation at a

higher, at the program level the high level at which the program was written line by line there because that is the users view of the process. Whereas, as for as the system is concerned, so this is working at the machine language level. So, for the system machine level debugging is easy. We must provide something at both the levels at the machine level as well as user level, so operating system may support like that.

Then communication, we may need to have communication between number of processes. So, this way we must have got virtual connections among processes, users and computer systems. So, virtual connection because there is no physical pipe or physical q that is implemented, all are virtual in terms of some memory, ultimately, they are all implemented by means of some memory locations. They allow users to send messages to one another screens browse web pages send email or login remotely transfer files from one machine to another. So, these are various communication type of system programs that are provided in the system. And for implementing them as I said that it requires some operation to be done in terms of system calls available with the operating system.

(Refer Slide Time: 05:23)

System Programs (Cont.)

- **Background Services**
 - Launch at boot time
 - Some for system startup, then terminate
 - Some from system boot to shutdown
 - Provide facilities like disk checking, process scheduling, error logging, printing
 - Run in user context not kernel context
 - Known as **services, subsystems, daemons**
- **Application programs**
 - Don't pertain to system
 - Run by users
 - Not typically considered part of OS
 - Launched by command line, mouse click, finger poke

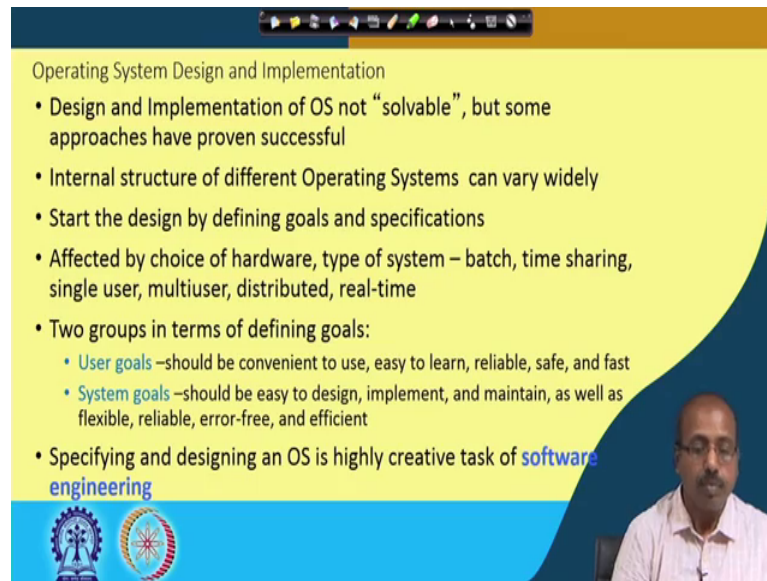
For background services; so maybe some of the services are to be launched at boot time, like system start-up and then terminate. And then some system some from system boot to shutdown, like we may like to have both of this service type of services available. Some services they will be invoked at the start-up time and then maybe when the system is getting up, and when the system is getting down like that. And then something that runs

for the entire time period of the system, like whenever the system boots that process comes up or the service comes up like provide facilities. Like disk checking process, scheduling, error logging printings. So, these are the different background services that goes on. So, as an user of the system, so you do not see those services, but those services are going on.

Run in user context and not kernel context. So, they are user level program. So, though they, so you can say in some sense that they are low priority user processes. So, they are running at the background gathering is information which will be used for by the system administrator later for improving the performance of the system or making a better usage of the system resources, but we can so they are not kernel process they are system mode process only. So, they are known as services sub systems or daemons. So, in if you are familiar with say Linux or Unix operating system. So, you will see that there are large number of daemons that are running; so this daemons are nothing, but this background services.

Then we have got application programs they do not pertain to system they are run by users and not typically consider part of OS. So, they are actually the for example, finding roots of a quadratic equation when we write the program, so that is of application program that is running s, or some database that is running, so that is also an application program that is running. So, they are launched by command line, mouse click or finger poke. So, whatever we do, so on a command line interface we give a command for execution on a graphical user interface we give a mouse click for invoking the program or if it is say a touch screen, so we can use our finger to start some service.

(Refer Slide Time: 07:49)



Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system – batch, time sharing, single user, multiuser, distributed, real-time
- Two groups in terms of defining goals:
 - User goals –should be convenient to use, easy to learn, reliable, safe, and fast
 - System goals –should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Specifying and designing an OS is highly creative task of **software engineering**

The slide features a yellow background with a blue wave on the right side. At the bottom left, there are two logos: one of a university and another of a circular emblem. A video inset in the bottom right corner shows a man with glasses and a light-colored shirt speaking.

Operating system design and implementation; so, design and implementation of OS is not solvable, but some approaches are proven successful. So, it is not that we can always ensure that my OS design will be the most efficient one that is possible, because many of the problems that we have in the domain of operating system. So, they belong to the NP hard problem, so they cannot be solved so easily, but the approaches they will try to make it more successful.

So, internal structure of different operating system can vary widely because they are designed by different groups and one group will not divulge the design to another group. So, as a result each group is trying in an independent fashion and they come up with different structures for the operating systems.

Start the design by defining goals and specifications. So, we tell what do you want to achieve and then we want to go step by step defining the system and going to the actual implementation.

They are affected by the choice of hardware type of system batch time sharing single user multi user distributed real time etcetera. So, on which hardware platform we are going to implement, so that a particular hardware will provide us certain features. So, like previously when this microprocessors or microcontrollers were being developed, so they are people, where looking for only higher and higher speed like that. But after that

people understood that, it is not sufficient to have the raw speed there, the operating system must be able to take help from those a piece of hardware those facilities that are that we are providing.

So, after that they started integrating several operating system related modules like the memory management unit, then the security things, so they have got they are getting introduced into the architecture itself or the hardware of the processor itself. So, it is also the waste design is also getting affected by the availability of those choices in the hardware. So, which hardware we are using, accordingly certain style of design will be followed. Also the type of the system that we want to design, whether it is a back system or it is a time sharing system, it is a single user or multiuser distributed system, whether it is a real time system or not, so that way there are several design issues that will come up and that will determine what should be the design paradigm.

Two groups in terms of defining goals, one is the user goals. User goals normally they say that it should be convenient to use easy to learn reliable safe and fast. A basic obstacle in going to a new operating system is that we do not know the commands there. Nowadays, with the advent of this GUI based interface so it has become more or less seems. So, even if I do not know the system commands, so I will be able to use the system to a good extend, only if I need some sophisticated feature of that system being used. So, I need to know the details of the system and what are the associated parameters and what are the associated commands like that.

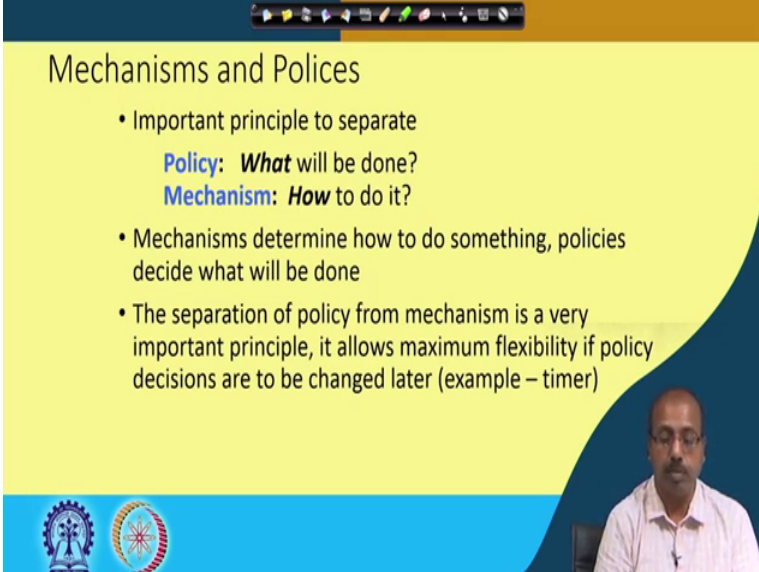
So, that requirement has pushed the systems towards GUI. Now, so that is the user goal, that is basically an user goal. On the other hand, the system goal it should be easy to design implement and maintain as well as flexible reliable, error free and efficient. So, it should be easy to design the system like whether you write the operating system in high level language or the assembly language that is a big choice. So, if I can write at high level language then expressing my concerns are much easier. So, in ensuring the constants writing down the constant in the program is easy.

So, that way how easy it is to design , how easy it is to implement. Maintain means tomorrow there is some report, something some bug report or some enhancement things to be done. So, how easy it is to modify it how easy it is to maintain. Then flexibility

reliability how much is the error, ok. So, these are the issues from the system point of view.

Specifying and designing an operating system is highly creative task of software engineering. So, that definitely the software engineering it will try to document the whole design process. So, documenting the OS design process is definitely difficult because OS itself is very complex and then documenting for that becomes more difficult.

(Refer Slide Time: 12:29)



Mechanisms and Policies

- Important principle to separate
 - Policy:** *What* will be done?
 - Mechanism:** *How* to do it?
- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

The slide features a yellow background with a blue footer. In the bottom right corner, there is a small video inset showing a man with glasses and a white shirt speaking. The footer contains two logos: one on the left and one on the right.

So, if you into the mechanisms and policies then the important principle to separate like policy means what to be done and mechanism is how to do it. So, policy and mechanism, policy wise we may adopt something, but when we come to the mechanism, so it may or may not be easy to do it. So, mechanisms determine how to do something, policies decide what will be done, ok. The separation of policy from mechanism is a very important principle it allows maximum flexibility if policy decisions are to be changed later; for example, the timer. So, it may so happen that we may want to introduce more accurate timers. So, so that is the policy decision.

Now, if the policy decision tells that, I want to use a better timer then I can go for changing the actual implementation part of it. So, if the policies is I should have flexibility, so that policy decisions can be changed easily later.

(Refer Slide Time: 13:33)

Implementation

- Much variation
 - Early Operating Systems were written in assembly language
 - Then with system programming languages like Algol, PL/1
 - Now C, C++
- Actually usually a mix of languages
 - Lowest levels in assembly
 - Main body in C
 - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- High-level language easier to **port** to other hardware
 - But slower
- **Emulation** can allow an OS to run on non-native hardware

On the implementation side there can be much variation like early operating systems. There are written in assembly language and then with system programming languages like Algol, PL 1 and now the most of them are written C, C plus plus like that. So, there are lot of variation in the OS designs.

So, when it comes to the high-level language the port needed becomes easy, the possibility of making error in writing the OS code become less, because of the very simple reason that this that a high-level the user has got better control of the program; whereas, if you are writing at the assembly level then possibility of making error in the logic is high.

So, ideally it should be a mix of the languages, for the lowest levels in assembly main body in C and some programs in C plus plus or scripting language like Perl, Python, shell scripts etcetera. So, the lowest level in assembly because when you are writing in the assembly, so you get the access to the registers CPU registers directly. So, particularly the basic input output services portion, so that is ideally written in terms of the assembly language of the underlying processor and second thing is that normally we want to optimise the performance of the system.

Now, if you want to optimise that the lowest level it should be implemented in a fashion, so that it is the fastest in execution. So, for fastest execution we must write in the

assembly language or which easily translated to machine code. So, this lowest level of the operating system is normally written in assembly, on top of that the main body is written in the C language and system programs may be written which is system programs are higher level than this main OS part, which are using this OS calls for getting the services done. So, they may be written even in high level language that is C, C plus plus, Perl, Python etcetera.

High level language they are easier to port to other hardware definitely because I can just compile the program at the destination machine and I get back the code. So, that way it is I get back the OS code, so that that is the reason that I should be able to take the program from one platform to another platform very easily. But definitely the program that we get will be slow, like assembly level language program when somebody is writing he is actually a master in that system.

So, natural he the program that will be written will be taking a care of this system in more detail, so that will be very efficient but in case of assembly and high-level language programs. So, it is actually done by some compiler and compiler may not be able to exploit the highest level of efficiency that the system can give. So, that way we have got flexibility as well as difficulty in using high level and machine level code.

We can have emulation that can allow and OS to run on a non-native hardware. So, emulation means a like if you have say, if you have a computer system, so that has got say, if you have got a computer system suppose this is a machine which is based on say Sun Solaris and there I am developing an operating system, so that will ultimately run in say I am developing an operating system for say Linux, some version of Linux I am modify.

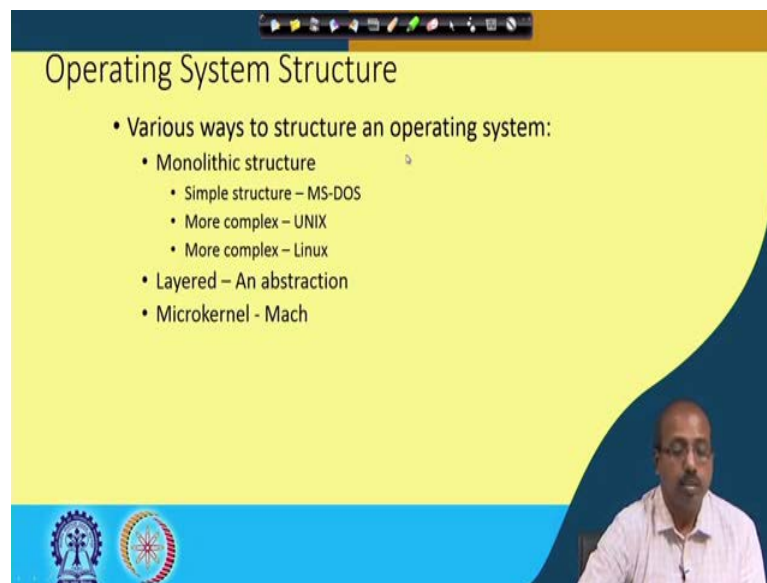
Now, you see that for running this Linux. So, here I have got some native platforms. So, on this may be running on some hardware say it may be some Pentium processor on to which it is running. Now, I can develop the code for Linux here, but this Linux maybe targeted to some machine which is a some processor which is from AMD. So, it may be running on an AMD processor.

But even if this Linux code is developed and it is checked on this, so I am not sure that whether it will be running on AMD or not; so for that purpose what I need to do is

somehow emulate the behaviour of AMD at this point. So, there is another layer of software that will be emulating the AMD hardware by using the Pentium hardware. So now, after developing the Linux program if I run it here so through this AMD interface then I can be sure of most of the operations of this new operating system that whether it will be run properly on the AMD platform also or not. So, we will be getting some confidence on that.

So, this is the process of emulation that one processor is emulated on another piece of hardware where the actual hardware is something different on which it is running. So, this emulation it can allow an OS to run on a non-native hardware. So, here this AMD is a non-native hardware and it is running on this Pentium processor, so that is via this emulation of AMD.

(Refer Slide Time: 18:59)



The slide is titled "Operating System Structure" and features a yellow background with a blue wave-like shape on the right side. At the top, there is a navigation bar with various icons. The main content is a bulleted list:

- Various ways to structure an operating system:
 - Monolithic structure
 - Simple structure – MS-DOS
 - More complex – UNIX
 - More complex – Linux
 - Layered – An abstraction
 - Microkernel - Mach

In the bottom right corner, there is a video inset showing a man with glasses and a white shirt speaking. At the bottom left, there are two logos: one of a gear and a person, and another of a circular emblem with a star.

Next we will have; so there are various ways to structure an operating system like we can have monolithic structure, we can have layered structure, we can a microkernel-based structure. So, monolithic structure means that entire operating system is a single piece of software. Like say simple structure like this MS-DOS. So, which our Microsoft disk operating system, which is which range the market for a long time, so that is one such monolithic piece of operating system.

So, we have got more complex like Unix operating system. So, that is also a monolithic one, it is also there similarly Linux operating system, so that is also a monolithic piece of operating system. Then we have got a layered structure also where operating system is developed in terms of layers.

So, in terms of the basic input output services on top of that we create one layer of the OS, on top of that we create another layer, so that way user seats at a much higher level and higher layer and this you can get service from the lower layers to get the job done. So, this is the layered approach, we can have this microkernel-based approach. For example, this mac OS it has got this microkernel, where this kernel functions will be utilised for getting the operations done.

(Refer Slide Time: 20:21)

MS-DOS

- MS-DOS – written to provide the most functionality in the least amount of space
- MS-DOS was limited by hardware functionality.
 - Not divided into modules
 - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

The diagram illustrates the layered architecture of MS-DOS, showing the following components from top to bottom:

- application program
- resident system program
- MS-DOS device drivers
- ROM BIOS device drivers

The slide also features two logos at the bottom left and a small video inset of a man in the bottom right corner.

So, looking into MS-DOS; MS-DOS return to provide the most functionality in the least amount of space. So, we have got their limited hardware facilities. So, node not divided into modules. And it has some structure its interface and levels of functionality and not well separated.

So, we have got this, at the lowest level we have got this ROM bios device drivers that can talk to the devices through this basic input output services. Then on top of that we have got dos device drivers that uses this bios device drivers to have some higher level of device drivers and then there are some resident system program, so that there we have

got the services that also. So, they are basically the portion that is providing the system call interfere the system sort of thing.

So, they are any application program running, so it will be using this resident system program for executing this for taking help of this MS-DOS device drivers and bios device drivers for getting the job by the underline hardware.

(Refer Slide Time: 21:29)

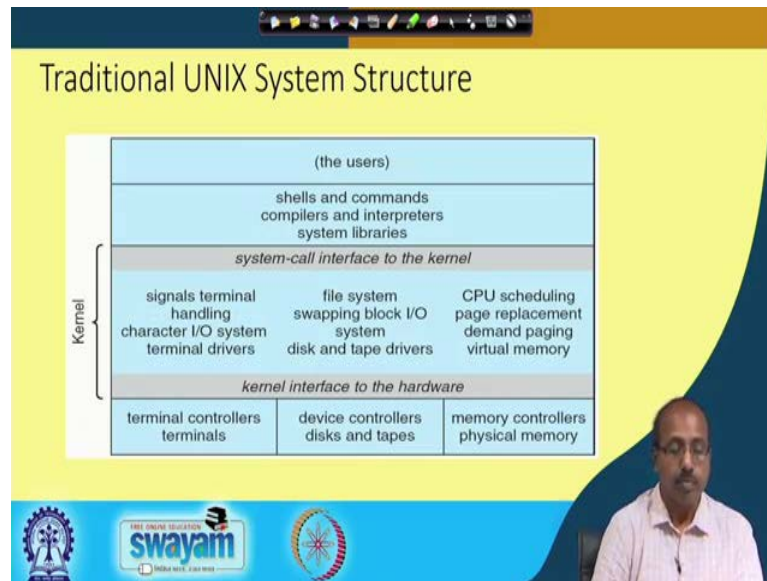
UNIX

- UNIX – the original UNIX operating system had limited structuring and was limited by hardware functionality.
- The UNIX OS consists of two separable parts
 - Systems programs
 - The kernel
 - Consists of everything below the system-call interface and above the physical hardware
 - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

Then in the Unix operating system the original Unix operating system had limited structuring and was limited by hardware functionality. The Unix OS it consists of two separable parts, one is the system programs and another is the kernel.

So, system programs we have seen like in the slash bin directory there are lots of such programs and there are they are actually the command that the Unix will support. And the kernel part, so kernel part it consists of everything below the system call interface and about the physical hardwares. So, it provides the file system CPU scheduling, memory management; so all those functions which are critical for the operating system, so they are put into the kernel. So, whenever we need to get a service from the system, so we have to make a system call or take help of some system program which in turn takes help of system call. So, ultimately all of them boil down to system calls and those system calls they will be executed and they gives us the service.

(Refer Slide Time: 22:31)



So, traditional Unix structure is like this. So, we have got a terminal controller terminal the control controller of our terminals, we have got device controllers for disk and tips, we have got memory controllers for physical memory. So, this is the lowest level. Then on top of that we have got the kernel level. So, kernel level it has got, this kernel interface to the hardware. So, this here we are actually you are having the device driver interfaces. So, this we are each of this device classes they will have their drivers and this kernel will be invoking those device drivers to get the job done by individual devices.

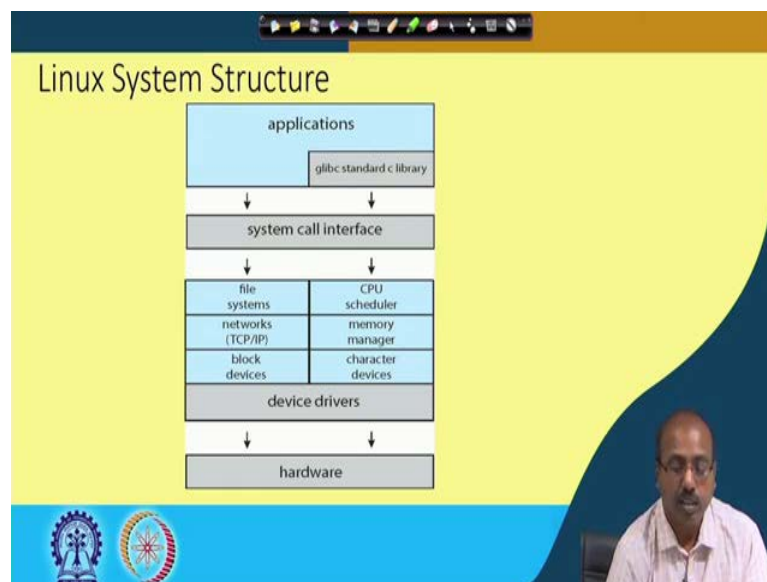
Then we have got this system call interface to the kernel and the kernel we have got several types of functions like signals terminal handling, character IO system terminal drivers. So, these are the different type of facilities that are provided for the terminal. So, it will be having routines for how to handle the terminal, how to send a stream of characters to the terminal and how to get a string of characters that from the terminal. So, like that both input and output.

Similarly, for file system, so we have got the calls by which we can get the disk and data from disk and tip. So, they will have the disk and tip drivers which will be doing those operations. So, or and similarly for the CPU scheduling part then this memory interface like page replacement demand paging virtual memory. So, they are all provided by and again by set of kernel services.

So, from the user level there will be system called interface that will take it to the kernel mode and in the kernel mode it will be doing all those operations. So, users at when they are sitting at this level. So, users can either give calls for can make this shell commands give the shell commands which will ultimately translate to system call for the system, so that is one possibility.

Other possibility is that the users can use the compilers and interpreters to compile their program written in high level language to some application program and that application program may be given for running. And when this application program is running it may in turn give rise to number of system calls and those system calls will be executed by the kernel. So, this way traditional Unix operating system, so they follow the structure.

(Refer Slide Time: 25:11)



On the other hand, this Linux system, so they actually I have got almost similar structure at the lowest level we have got the hardware, on top of that we have got device drivers. Devices are broadly divided into block devices and character devices. Block devices they are actually the devices were like that tape, disk etcetera where we have we have talking in terms of blocks, that is we do not do single character data transfer it is in terms of blocks of 1024 bytes or 4096 bytes like that.

And character devices are like the like the display then your keyboard like that, so they are characters are the transfers in terms of characters. So, we have got character device

handlers, we have got block device handlers, we are the memory manager is their which will be handling the memory internal memory allocation and de allocation part. Then we have got this network manager, so that will be TCP that will follow this TCP-IP protocol for this file transfer data transfer etcetera.

So, we have got file systems related module, so which will be doing this, which will be implementing file system in terms of creating new file opening file etcetera. Then the CPU scheduler which will be determined like which job will be or which process will be getting the CPU next for execution, so that is why CPU scheduler.

So, at the top level we have got the applications and this we are there is a standard C library is a part of this Linux operating system, and then this Linux operating system so it a application is running. So, whenever it needs a service from the system. So, it will be making this it will be going through this system call interface and from there it will be coming to this kernel mode where it will be doing this different operation whichever is requested for.

(Refer Slide Time: 27:05)

Modularity

- The monolithic approach results in a situation where changes to one part of the system can have wide-ranging effects to other parts.
- Alternatively, we could design system where the operating system is divided into separate, smaller components that have specific and limited functionality. The sum of all these components comprises the kernel.
- Advantage: Changes in one component only affect that component, and no others, allowing system implementers more freedom when changing the inner workings of the system and in creating modular operating systems.

The diagram shows a box divided into two sections: 'Interface' and 'Impl'. Above the box are two circles, one with a minus sign and one with a plus sign.

So, modularity if you look into, the monolithic approach results in a situation where changes to one part of the system can have wide ranging effects on other parts. So, since whole thing is monolithic, so if you change somewhere. So, maybe as a cascaded effect, so you need to change at many places, so that is the problem. Alternatively, we can

design the system where operating system is divided into separate smaller components that has specific and limited functionality and sum of all these components comprises the kernel.

So, this is a standard method that is followed now in a new systems because now the whole job is divided into modules and it is a file system modules there is something some augmentation as to be done it should not affect my CPU scheduling module. So, like that we have this type of separation, so this will help us. So, in our other development also any software development you know that this modularity helps a lot, the same is true for the operating system also.

So, change advantage we get, like changes in one system or one component to only effect that component and no others, allowing system implementers more freedom when changing from, the changing the inner workings of the system and in creating modular operating system. So, basically the things that we are dividing them into two parts, one is the some part of the depending upon the operation.

So, we are dividing it into two parts like one part is the interface. So, this is the interface part and other is the implementation, the implementation part. So, if you change the interface part then implementation may not change, similarly you may change the implementation part keeping the interface fixed. So, that is one thing that we have, that is one advantage of modularity.

Second thing is that there are different modules in my system. So, if this module is undergoing some change. So, the other module they did they did not change. So, they remain as it is. So, this way this modularity helps us. So, instead of having monolithic piece of operating system so this later development. So, they are making it distributed thing like they are making it divided into a number of module, so that job is distributed among it number of modules and the different setup developers can work on different modules.

Many of the operating systems today the development is contributed by groups spread over all over the world. So, if some group is concentrating only on the file management portion. So, they need not be bothered about the CPU scheduling part and that way if some augmentation is done in the file management portion, so that can be easily

integrated with the existing system. So, that is a way this operating system design is now growing. So, this modularity is very important there.

So, we will continue with this in the next class.