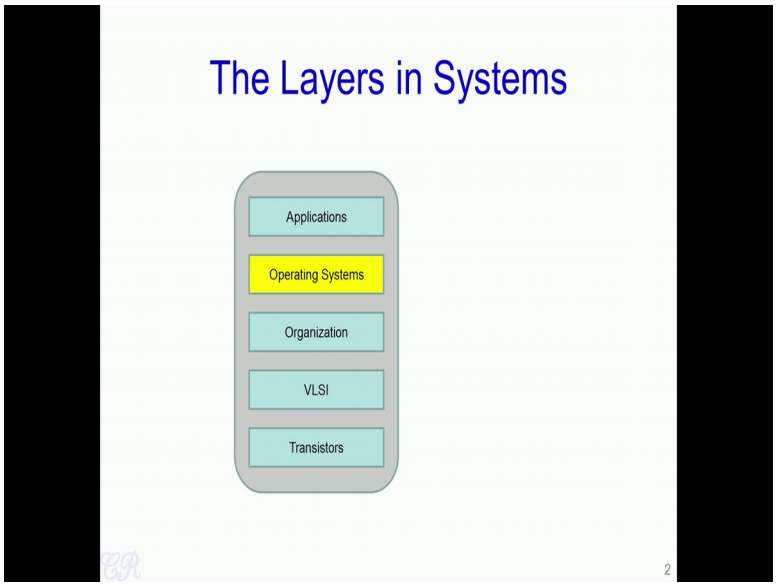
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 01
Lecture - 01
Operating Systems (An Introduction)

Hello and welcome to the first weeks lecture for the course An Introduction to Operating Systems. This week we will be building a platform for the course upon which the following weeks lectures will depend upon. In this particular lecture, we will give a very brief introduction to OS. In particular, we will look at where the operating system actually fits in the entire computer and also we will see what is the essential role of the operating systems in the computer.

(Refer Slide Time: 00:57)

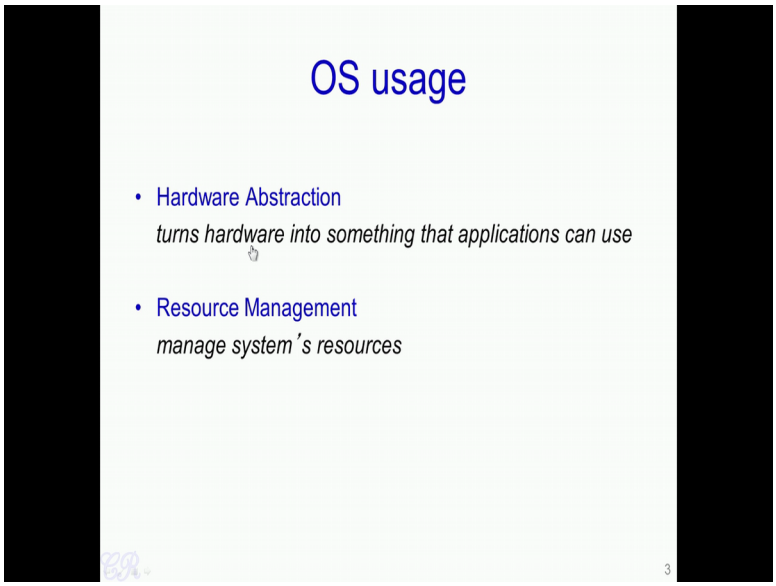


So, when we look at computer systems, we can think of it in different layers. So, right at the bottom layer are millions and billions of Transistors. So, these Transistors are typically CMOS Transistors and these Transistors are then composed together to build several logical Gates and these Gates would include several digital logical Gates, such as the AND gate, OR, XOR and so on. Included in this particular layer are various things like the Memory cells, Flip Flops, Registers and so on.

Now, all these basic VLSI units are then organized into various forms to build things like the Memory, RAM, the Decode unit, the Instruction fetch unit and so on. So, this Organization (in the slide above) which is the third layer in the computer system is what we actually see as the Hardware. So, this is the Physical Hardware that we actually purchase from the store. Now, when we purchase this Computer Hardware, we could execute several different Applications. For instance, we could have Office, Applications or Internet Explorers and so on.

Now, sitting between these Applications as well as the Hardware is the Operating Systems. So, the Operating System essentially would manage both the Applications that execute on the Computer as well as it would manage how the Resources are utilized in the system. So, let us see more in detail how the Operating Systems are used in the entire scheme.

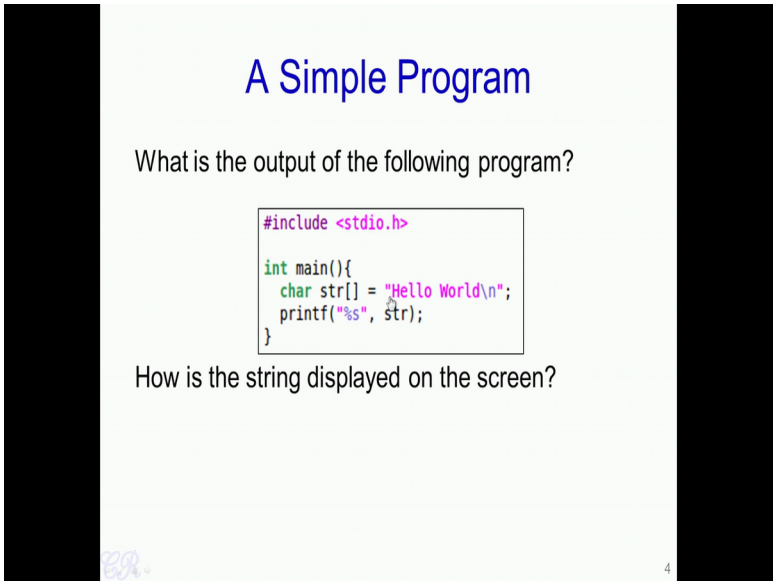
(Refer Slide Time: 02:51)



To look at it broadly, the Operating System is used for two things. It first provides Hardware Abstraction and second manages Resources in the system. The Hardware Abstraction essentially is used in order to turn the Hardware into something that Software Applications can utilize easily, while the Resource Management is required because of the limited resources that are present in the Computer.

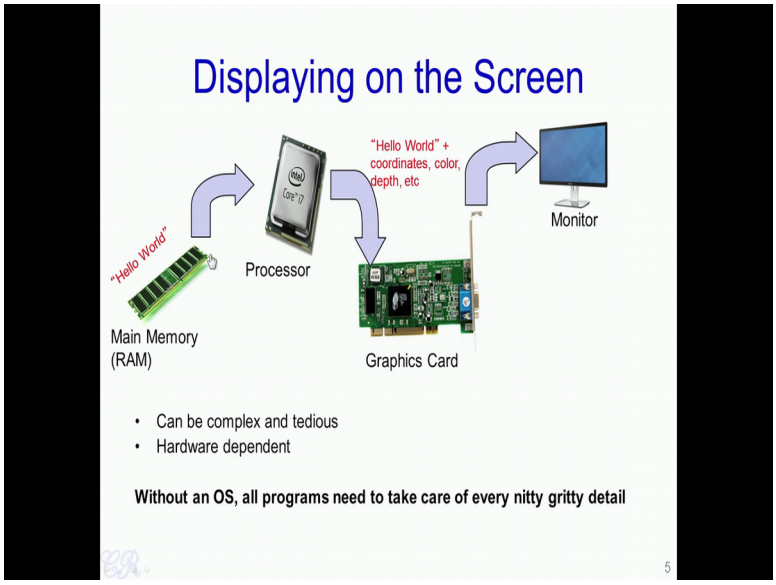
So, we will look at these two uses of the Operating System in more detail.

(Refer Slide Time: 03:31)



Let us start with this very simple program. So, this is a program written in the C language and essentially it is going to print the string “Hello World” on to the monitor. So, this is a very simple program and essentially the string “Hello World” is stored in memory and it is pointed to by this pointer str[ ]. Now, printf (mentioned in above slide) is passed this pointer str and would result in the string being printed on to the monitor. Now, the question which comes is how exactly is the string displayed on to the monitor? What is the process involved to get the string, which is stored in memory to be displayed on to the monitor?

(Refer Slide Time: 04:25)



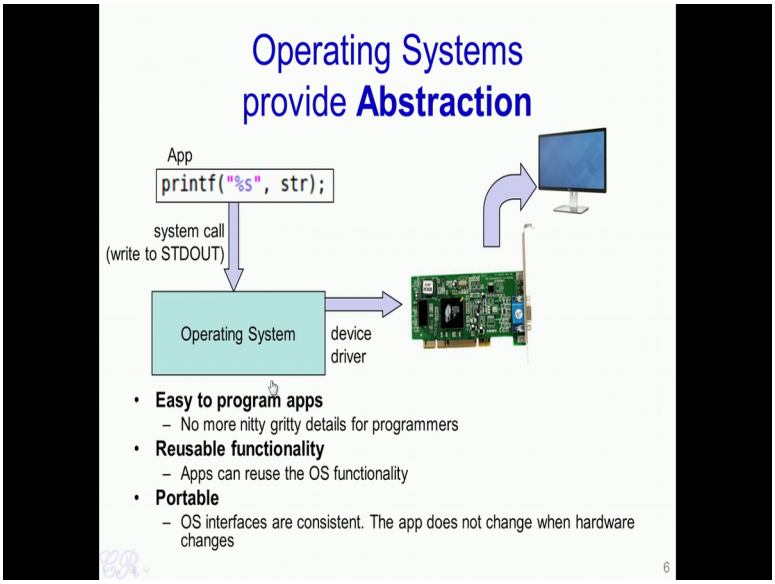
So, in fact when we look at the entire scheme, the string “Hello World” would be present in some memory location in the Main Memory which is also called as the RAM. Now, there are certain Instructions in the Processor would read the String byte by byte from the Main Memory into Registers and then copy them on to something known as a Video Buffer. Along with copying the String “Hello World” to the Video Buffer, other attributes are added. For instance, things like the color of the string to be displayed, the x y coordinates on the monitor where the “Hello World” string needs to be displayed and other monitor specific attributes such as the depth and so on.

Now, this string which is copied to the Video Buffer is then read by the Graphics Card which would then display it on the Monitor. So, this is the entire process of displaying a string “Hello World” on to a Monitor. So, now you would see that doing this is not trivial. It is in fact quite complex as well as tedious. Imagine that every program that you write would require to do all these things like knowing where in the memory the “Hello World” is stored and then, how to actually display it on to the Monitor, how to compute the coordinates, how to specify the color, the depth and other attributes and how exactly to pass this information to the Graphics Card and so on.

Another aspect is that this is extremely hardware dependent. Any change in one of these

things. For example, if the Processor changes or if the Graphics Card or Monitor changes then it is quite likely that the Program will not work. For example, if the Monitor changes, then there may be certain attributes which need to be specified differently for the new Monitor or if the Graphics Card changes, then perhaps the way the coordinates are set where the depth and color are set for this string “Hello World” would differ. Therefore, without the Operating System, every programmer would need to know about the nitty-gritty details about the Hardware. Essentially he would need to know what Hardware is used in the System and also, how the various aspects about the Hardware fit with each other.

(Refer Slide Time: 07:15)



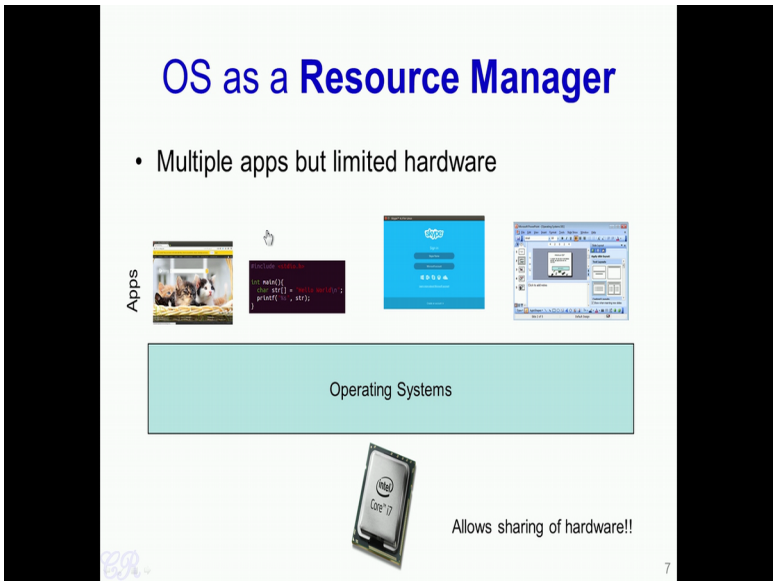
So, Operating Systems essentially provide Abstraction. They sit in between the Applications that we write and the Hardware, and abstract all the nitty-gritty details of the Hardware from the Applications. So, a simple statement such as this printf(“%s”, str) would eventually reside in something known as a system call which would trigger the Operating System to execute. The OS will then manage how the String str will be displayed on to the Monitor. Everything such as how the color needs to be set, how the x y coordinates need to be set and so on would be done internally by the Operating System. So, from an Application perspective and from a programmer perspective, all these details are abstracted out.

So, as you would see this would make writing programs extremely easy. So, the programmer need not know the nitty-gritty details about the Hardware any more. A second advantage is the Reusable functionality. Essentially Applications can reuse the Operating System functionality. So, what we mean by this is that since the OS abstracts the Hardware details from the Applications, all Applications that execute in this system could just reuse the Operating Systems features.

For example, every Application that uses printf will be invoking the Operating System and the OS will then take care of communicating with the Hardware. There is the single module in the Operating System which handles all printf's from all Applications executing on the System. A third advantage is the Portability. Essentially what this means is that when we write a program which uses something like a printf statement, we do not really bother about what Hardware it runs on. It could run in a Desktop for instance or a Server or a Laptop or if complied appropriately, also in several embedded devices.

So, the underlying Operating Systems actually would then distinguish between the various Hardware, that is present and with then, ensure that printf would be executed appropriately depending on the Hardware. So, what we achieve with Portability is that essentially Applications will not change even though the underlying Hardware changes.

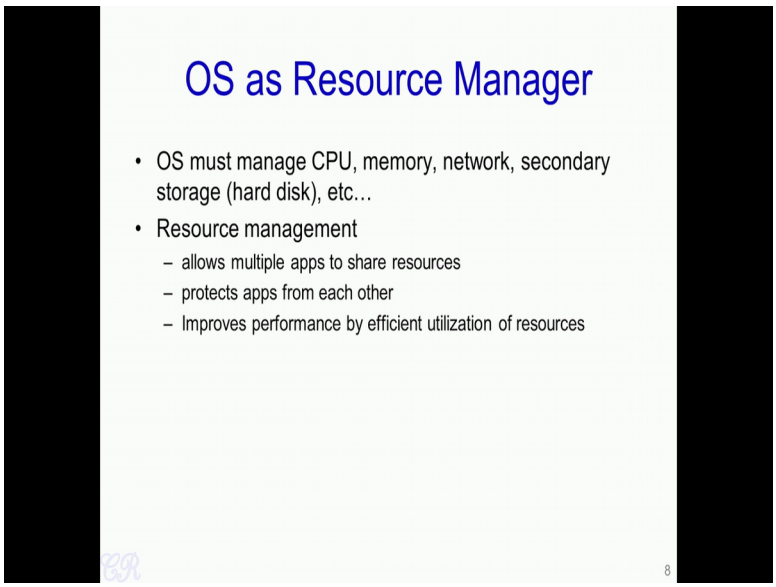
(Refer Slide Time: 10:12)



The second use of the Operating System is as a Resource Manager. In the Desktops and Laptops that we use today, there are several applications which run almost concurrently. For instance, we would be using a web browser to browse the internet and almost at the same time, we would be compiling some of the programs that we are written and are also executing them or we could be using Skype or a Powerpoint application at almost the same time.

Now, the fact is that we could have several applications running on a system, but the underlying hardware is constrained. Essentially, we have just one hardware which needs to cater to several applications almost concurrently. So, the Operating System ensures that it is feasible for multiple applications to share the same hardware.

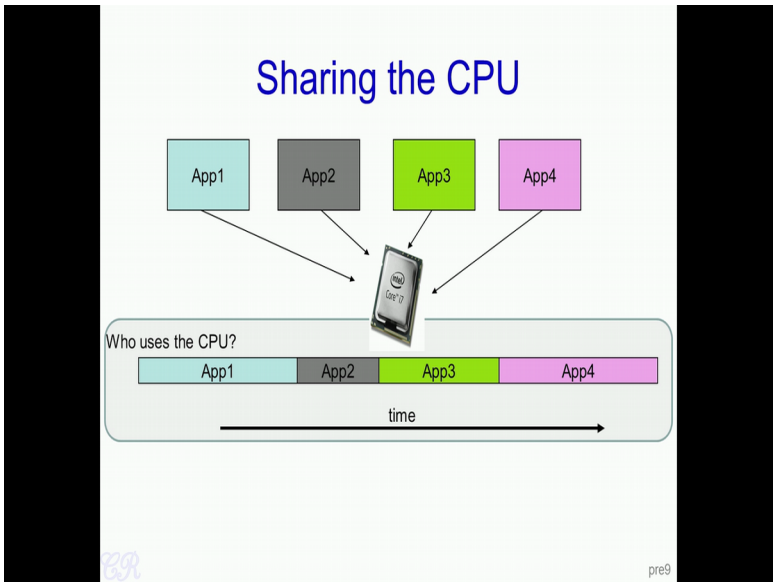
(Refer Slide Time: 11:15)



Now, within this Computer hardware, there are several components which the Operating System manages. For example, the CPU, the memory, network, the secondary storage devices like the hard disk, the monitors and so on. So, all these components within your computer have a restricted amount. So, you may typically have around 2 or 4 or 8 CPUs present in your system, also the memory may be restricted to 4 or 8 GB. You have typically one network card or one or two hard disk and so on.

So, the Operating System needs to manage all these various devices and components present in the system and share these components among several applications almost concurrently. So, with the help of the Operating Systems, it would be possible that multiple applications share this limited resources and also, the OS is built in such a way that applications are protected from each other. Essentially the underlying where the Operating System is designed is such that every component in the system is adequately utilized.

(Refer Slide Time: 12:35)

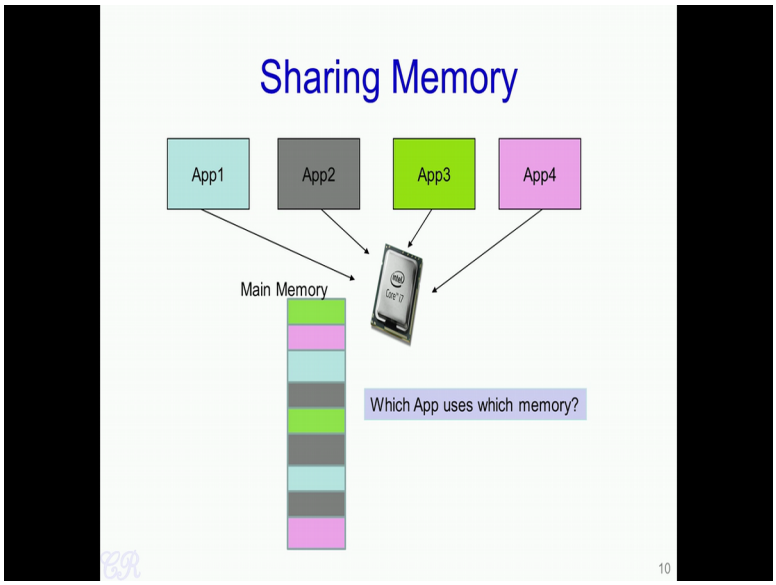


To take an example let us consider the CPU. So, systems typically would have one or two CPUs and multiple applications executing on that CPU. So, how does the Operating System share the single CPU among multiple applications? So, one way which was typically done in the earlier operating systems around the late 70's and early 80's was to allow one application to execute on the CPU till it completes and then, start the next application. For example, Application 1 is made to execute on the processor and after Application 1 completes its execution, only then Application 2 is made to start.

Now, this scheduling of the various applications on the processor is managed by the OS. Now, this scheme of sequentially executing one App after the other completes although very simple to implement in the operating system, it is not the most efficient way to do

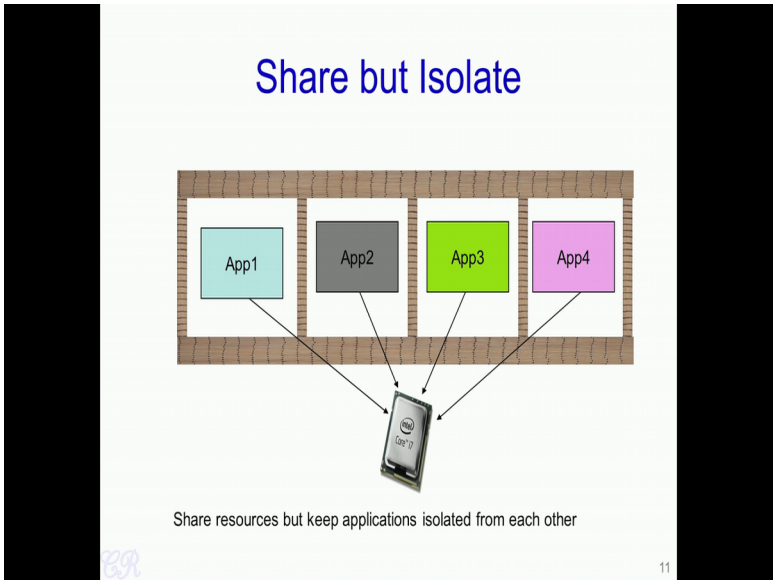
things. So, as we will see in a later video, we will see what are the issues with having Applications execute in this particular way and we will see how modern day Operating Systems manage to utilize this processor in a much more efficient manner.

(Refer Slide Time: 14:13)



An other important component in the computer system that requires to be shared among the various applications almost concurrently is the Main Memory. Now, in order to execute each of these applications needs to be present in the Main Memory that is the RAM of the system. The Operating System needs to ensure how this limited RAM resource is shared among the various applications executing on the system.

(Refer Slide Time: 14:46)



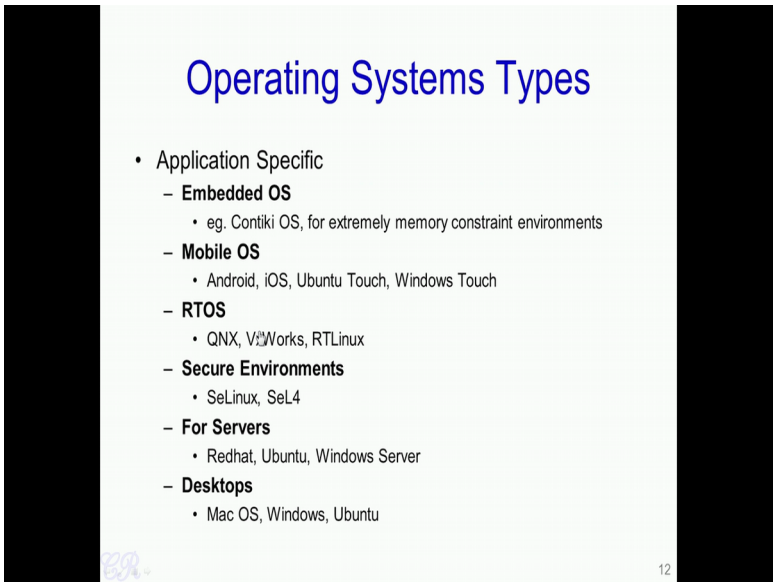
Now, what makes it difficult for the Operating System is that in spite of sharing the limited hardware resources among the various applications which are executing on the computer system, the sharing should be done in such a way so that applications are isolated from each other. So, each application the OS should ensure runs in a sand boxed environment that is Application1 should not know anything about Application 2, Application 2 should not know anything about the other applications running on the system and so on.

So, why is this isolation required? To take an example of why isolation is required, let us consider that Application 1 is a web browser in which you are doing a banking transaction. For example, you are entering your passwords and credit card details in the web browser which is executing as Application 1.

On the other hand, Application 2 may be a Gaming software and let us assume that it is having a virus that is it is a malicious application. Now, assume that we do not have Isolation. Application 2 would be able to determine what application 1 is doing or in other words, the malicious application will be able to determine what is happening in the web browser and therefore, may be able to steal certain sensitive information such as your passwords and your credit card numbers. Therefore, the Operating System should

ensure that all these applications are isolated from each other and as we can see this is not a very easy thing to do.

(Refer Slide Time: 16:45)



Operating systems are ubiquitous.. Almost every smart device that we use today has some form of Operating System present in it. So, this particular slide (mentioned above) shows the various classifications of operating systems depending on the type of application they are intended for. Needless to say each of these operating systems are designed keeping the Application in mind.

For example, the embedded OS such as Contiki operating system or Contiki OS are designed for memory constraint environments, such as wireless sensor nodes that are used in the internet of things. Operating System like the Contiki OS are designed with the power consumption kept in mind. So, these operating systems manage the various resources and also abstract hardware in such a way that the power consumed by the entire system is kept minimum. A second class of Operating Systems which many of you may be familiar with is the mobile operating system or mobile OS.

So, examples of these are the Android, iOS, Ubuntu Touch and Windows Touch. So, these operating systems like the Embedded OS are designed in order to ensure that the

power consumed by the device is kept minimum. So for example, these operating systems are designed, so that the battery charge of your mobile phone is extended for the maximum amount of time. So, the mobile OS's like the once we mentioned over here has quite similarities in this aspect with the embedded operating systems like the Contiki OS.

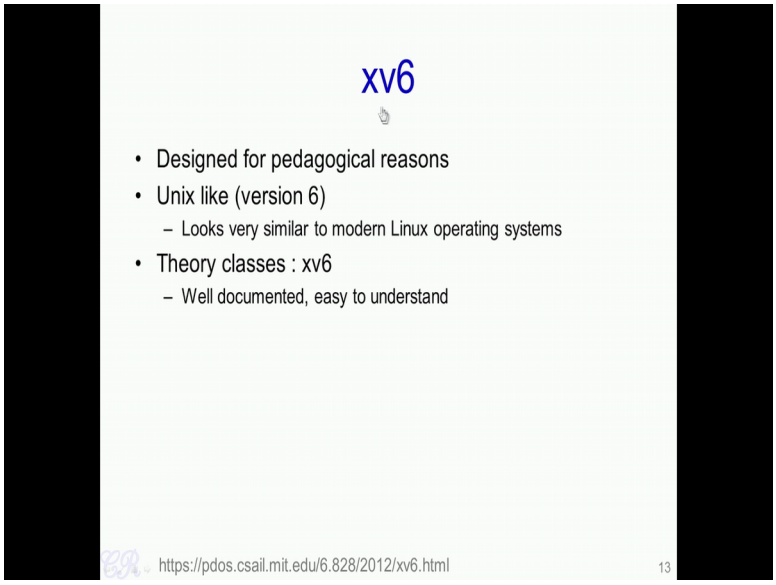
However, mobile operating systems unlike the Embedded OS's also need to take care of certain special devices. For instance the monitors or the LCD screens and key pads. In other words, the mobile OS also has user interaction which typically is not present in the Embedded OS.

A third type of operating system is the RTOS or Real Time Operating Systems. So, examples of these are QNX, VxWorks and RT Linux. So, these operating systems are used in machine critical applications where time is very important. For example, in several machine critical applications like rockets, automobiles or nuclear plants, a delay in doing a particular operation by the computer system would be catastrophic. So, these RTOS's are designed in such a way that every critical operation on the system is guaranteed to complete within the specified time.

Another classification of operating systems are those used in Secure Environments. So, examples of these are SeLinux and SeL4. So, these operating systems are especially utilized for applications where security is extremely critical.

So, these for example could be for web servers that host banking software and so on. So, the other classes of operating systems which you are quite familiar with are for those used for Servers and Desktops, such as the Redhat, Ubuntu and Windows Server OS's while desktop operating systems are for example Mac, Windows and Ubuntu. So, while these two operating systems have several features which are similar, there may be certain differences in the way the OS manages the various applications running on it.

(Refer Slide Time: 20:54)



The operating system that we will be studying for this course is the xv6 OS which is designed by MIT specifically for teaching purposes. So, the xv6 OS is small, well documented and easy to understand. Further, xv6 is designed to look similar to UNIX (version 6). So, what this means is that the way xv6 is designed is how various UNIX like operating systems like Linux actually works. Therefore, understanding xv6 would give you a nice insight about other modern day operating systems such as Linux.

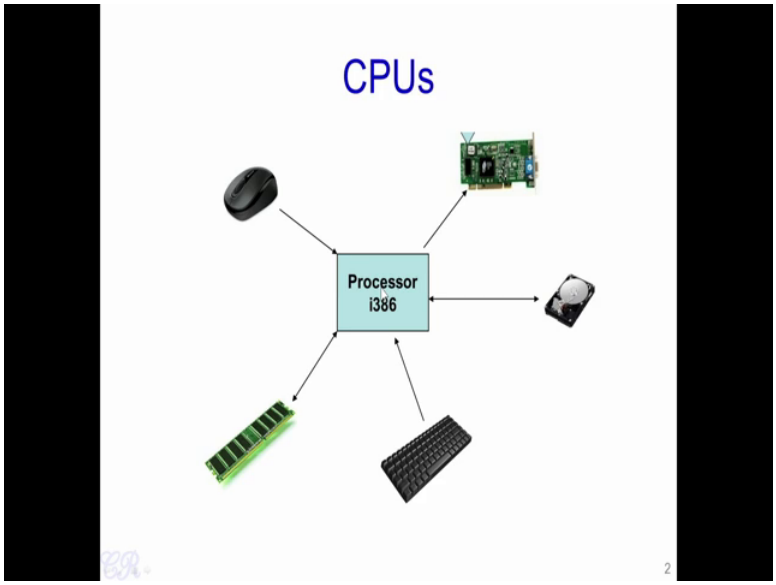
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 01
Lecture – 02
PC Hardware

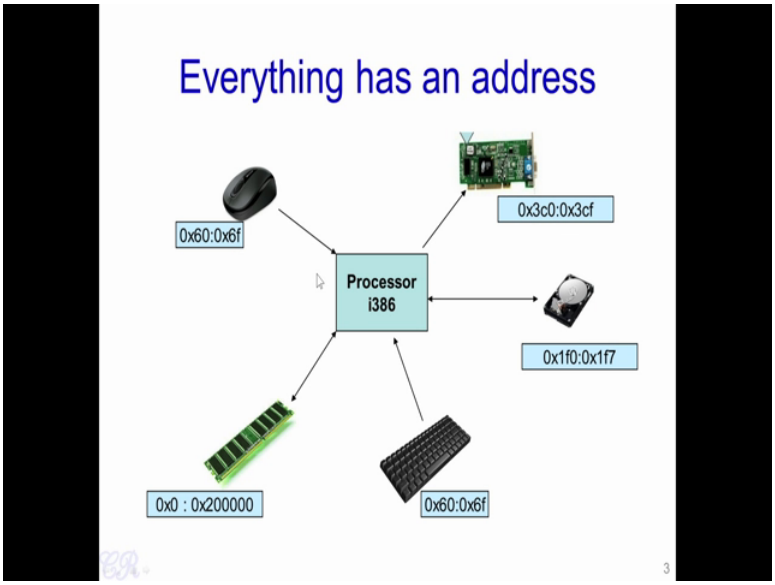
Hello. A lot of how the operating system works depends on the underlined hardware. Essentially, as we seen in the previous video, the Operating System is in charge of managing the hardware. So, before we go any further with, how an operating system works and how the OS functions? Let us take a brief background about the PC hardware. Essentially we will look at how a computer hardware is designed.

(Refer Slide Time: 00:50)



So, we know that the heart of any system is the CPU; and the CPU is interfaced to several devices such as the VGA card, a hard disk, a keyboard, RAM, mouse and so on. Now, in order that all these devices work with a single processor - what is required, is addresses.

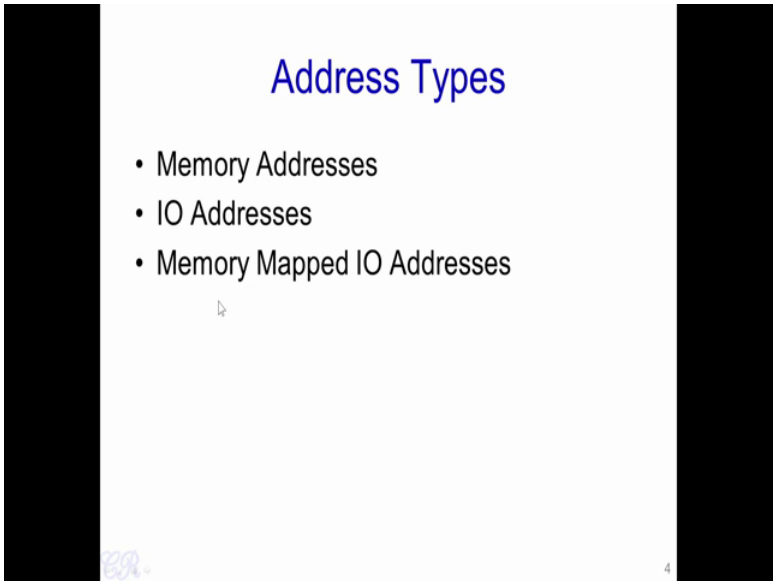
(Refer Slide Time: 01:19)



Essentially, each device in the system would have a unique address. And it is ensured that no two devices in the system would have the same address. For instance, we would have the hard disk which is present from the address range 0x1f0 to 0x1f7. So, what this means is that when the processor sends out an address on its address Bus, the hard disk would identify that the address is within this particular range and therefore, it will respond.

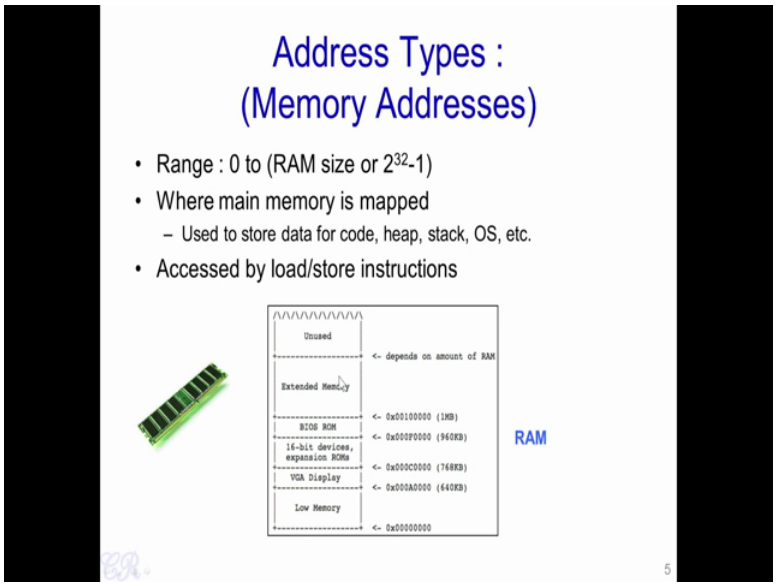
All other devices will then ignore that particular transfer on the processor Bus, because it does not fall in its particular address range. For example, if the processor puts out the address 1f2, then only the hard disk will respond. Because 0x1f2 falls within this particular range of the hard disk. So, if you look at something else like the mouse, which has address range of 0x60 to 0x6f, it is not going to respond to the processor's request.

(Refer Slide Time: 02:47)



Now in systems, there are 3 types of addressing which are generally used. One is called the Memory Addressing, next is the IO Addresses, and the third is the Memory Mapped IO Addresses. So, we will look at each of these Types of Addresses more in detail.

(Refer Slide Time: 03:10)



So, Let us start with the memory addresses. So, the memory addresses and most systems

would have a large amount of such memory addresses correspond to addressing the RAM. Each memory unit in the RAM is given a unique address. For instance, a RAM in a typical Intel system with 32 bits could be of size at most 2^{32} that is a Intel system with a 32 bit processor could have at most 2^{32}or 4 GB of RAM. So, each and every memory unit in this RAM has a unique address. So, this memory unit could be as small as a single byte, or in some systems it could be a word that is it could be of 16 bits or 32 bits.

Now, how is this RAM used? So, essentially the RAM is configured in this particular way as shown in the figure above, and this configuration is especially for IBM PC compatible Intel machines. So, this RAM as I mentioned has addresses to access each part of the RAM and the RAM could be as large as 4 GB. The addresses from 0x0 to 640 KB represented in hexadecimal as A0000 is known as the low memory. So, this particular memory was used in legacy computers like the 8086, 8286 and so on so.

So, the old operating systems like MS DOS would use this particular low memory. So, above this low memory from 640 KB to 768 KB, all addresses would pertain to the VGA display. So, again this is an legacy issue, where this particular area in the memory would correspond to the VGA display. This means to say that any ASCII characters placed in this particular area would then be taken by the video card and display it on to the screen.

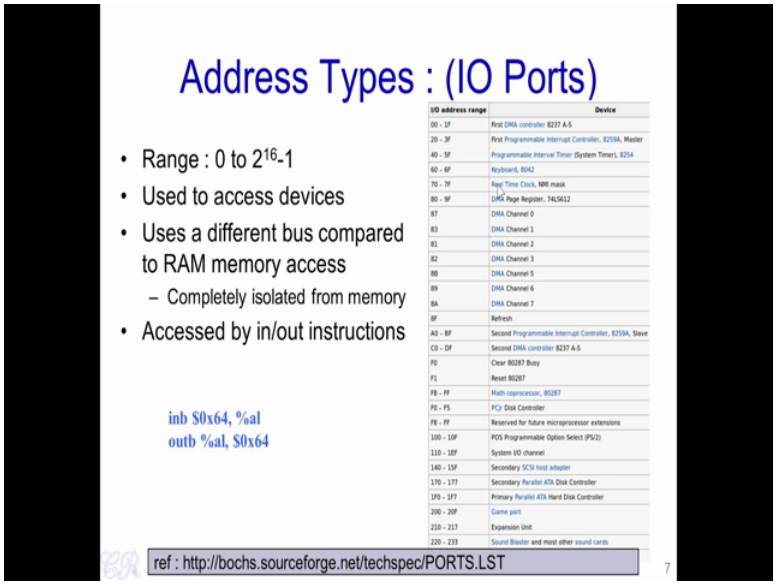
The region from 768 KB to 960 KB was known as the 16 bit expansion ROMs, so these also were legacy aspects present in legacy computers. And it is pertaining to every device that is used. Essentially, in the previous generations of computers in the early 80's to early 90's, the devices that were attached to the computer could have what was known as expansion ROMs, ROMs are the read only memory. And these ROMs could be accessed or rather these ROMs could be addressed within this particular memory region. Now what is important for us and what we still use in the present Intel systems that we use for desktops and laptops is this particular region from 960 KB to 1 MB. So, this was where the BIOS reside.

So, as many of you would know, BIOS is the basic input output system; and it is a read only memory which is present on your system. And it ensures that your system boots correctly and it also ensures that the operating system gets loaded. So, the area of the

RAM, essentially all memory addresses below 960 KB is typically not used in modern day systems. While the area from 960 to 1 MB that is 960 KB to 1 MB is used by the BIOS and only used during the booting of the system. What is actually used in modern day system is the RAM above this 1 MB region that is starting from what was known as the extended memory. And this extended memory could go as far as the amount of RAM is present in the system.

For instance, if you have 4 GB of RAM or 8 or 16 GB of RAM then this extended memory will extend up till that particular region. The parts or the addresses above this RAM area will be generally unused. So, as we will see in the later classes, in this particular course, we will see how the operating system loads itself starting from this 1 MB region. We would also see how the BIOS is going to be used to boot the operating system. Now, as we know this particular RAM is also used to contain various aspects of the applications that we execute. For instance, the code segment or the instructions present in the program that you execute, the heap, the stack as well as the operating system reside in this particular memory. Essentially all of them will reside above this extended memory part.

(Refer Slide Time: 09:41)



The next type of addressing is the IO addresses. Essentially in legacy computers like the

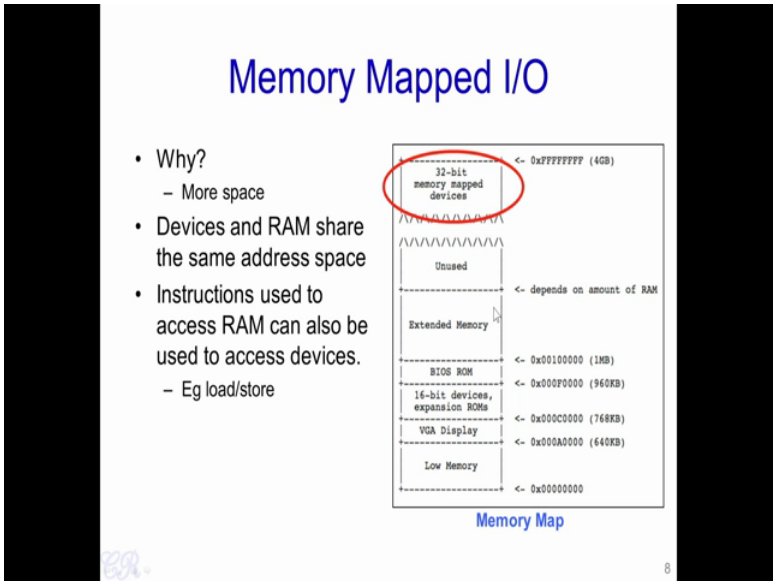
8086 and the 8088 and 8286, there was a separate memory region for devices. IO devices like keyboards, mice, hard disks and other programmable devices like a programmable interrupt controller could be mapped. So, unlike the previous addressing that we seen that is the memory addressing, so the IO addressing could be from 0 to just 2^{16} - 1. So, this is roughly around 64 KB and the IBM PC standard would define what the use of certain addresses are.

So, the IBM PC standard is a standard which was followed to develop the systems like especially the desktop systems and certain servers and these standard mentioned what devices should be present in what IO addresses. For instance, it mention that the keyboard should be present between 60 to 6F. So, the IO address - 60 to 6F.

In a similar way, the DMA controller would be present from C0 to DF. So, the other things like SCSI or a hard disk - a primary hard disk present in your system will be present from 1f0 to 1f7. So, in this way, several devices had very specific addresses in the system. So, what is the use of having such a specification is that we obtain compatibility. So, even now when you boot your laptop or your desktop PC, which is an Intel system or an AMD system, the BIOS will expect several things like it will expect that there is a keyboard which is connected to your system and the keyboard is present or rather the keyboard is accessible from the address range 60 to 6F. Similarly, it would expect other things like a programmable interrupt controller to be present from the IO addresses 20 to 3F.

Essentially, what is maintained even in the modern systems is the backward compatibility to the old computers that we used. So, the memory addressing as we are specified over here, even though a lot of it is legacy and was used quite a bit in the early 80's and 90's, even now since Intel and IBM followed the backward compatibility. A lot of the systems that we use for our laptops and desktops still follow this particular hierarchy. Now one problem, which was noticed with this type of IO addressing, was the limited amount of address range that was permitted. So, for instance over here we had at most 64 KB of addresses that could be present. So, this means that the number of devices that could be connected to this system through the IO addressing was limited.

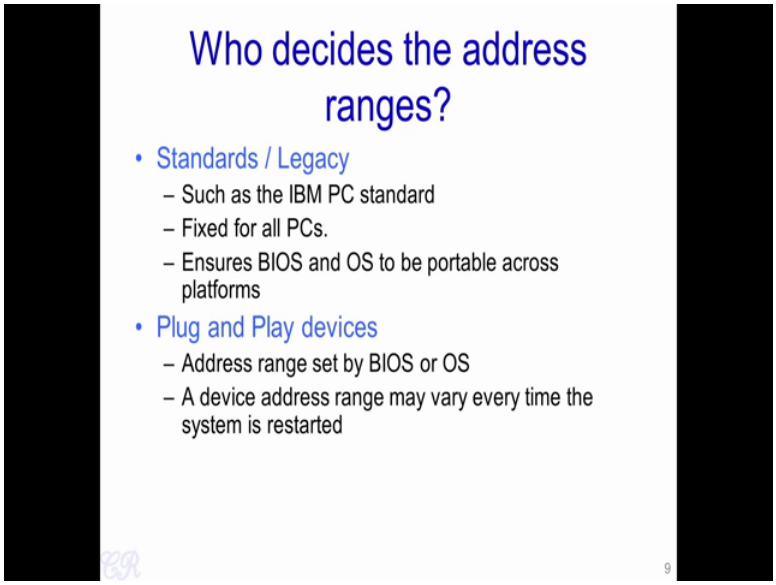
(Refer Slide Time: 13:53)



In order to extend this, what was used was known as the memory mapped IO. In this particular memory mapped IO, hardware devices would then connect or would be then mapped into regions in the memory addressing itself. So, as we have seen before, the RAM or the random access memory of the system would be addressed from the location 0 and extend upwards until the end of the RAM.

So, typically in the systems of the early and late 90's, we would have something like 256 MB, 512 MB or at most 1 GB of RAM. So, the space above this was unused. So, what devices would do, is they would map a certain region in this particular part of memory into their devices. What this means is that when the processor generates an address in this upper region (marked with red circle) of memory that is a high address in this upper region of memory then that particular address would be directed to the specific device and not to the RAM.

(Refer Slide Time: 15:13)



So, we have seen various address ranges that have been allocated to RAM such as the low memory region, the location where the BIOS should be present, the location where various devices like the keyboard, hard disk and so on should be present. Now who decides these particular address ranges? Essentially these address ranges have been decided by standards and a lot is impacted by legacy computers of the 80's and 90's. So, one very famous standard was the IBM PC standard. Essentially, this particular standard was fixed for all PCs, that is all personal computers right up from the 80's would be compatible with IBM PCs standard.

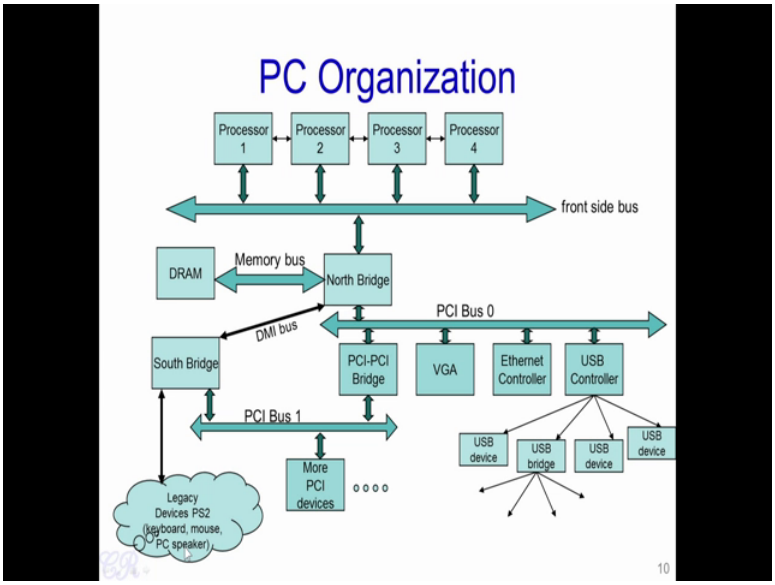
Again even many of the systems that we use today that is our Intel and AMD laptops as well as desktops are backward compatible to this IBM PC standard. So, the reason for this IBM PC standard or for that matter why a standard was required, was to ensure that the BIOS and operating system is portable across platforms.

So, for instance if I would write up an operating system today, my operating system would know exactly where the keyboard should be present irrespective to one of what IBM PC compatible system it is going to be loaded on. So, I could make a lot of assumptions about where various devices are present in the system. So, having such a standard would essentially ensure that the software that we write especially the software

that interacts with the hardware such as the BIOS and the operating system will be portable across several different platforms.

The other way in which addresses are decided upon is by something known as plug and play. So, plug and play devices do not have any fixed address location as we have seen for things like the keyboard or VGA, memory and so on. But rather, when the system boots and when the BIOS begins to execute, the BIOS will then detect the presence of some particular hardware in the system. For instance, the BIOS would detect that the system has a network card present; it would then ask the network card as to how much memory it requires and what type of memory it requires, whether it is a IO memory or a memory address. So, based on this, the BIOS will then allocate a portion of memory for that particular network card. So, the allocation would be fixed for each boot of the PC, but on the other hand the location in the address map would vary across each boot.

(Refer Slide Time: 18:38)



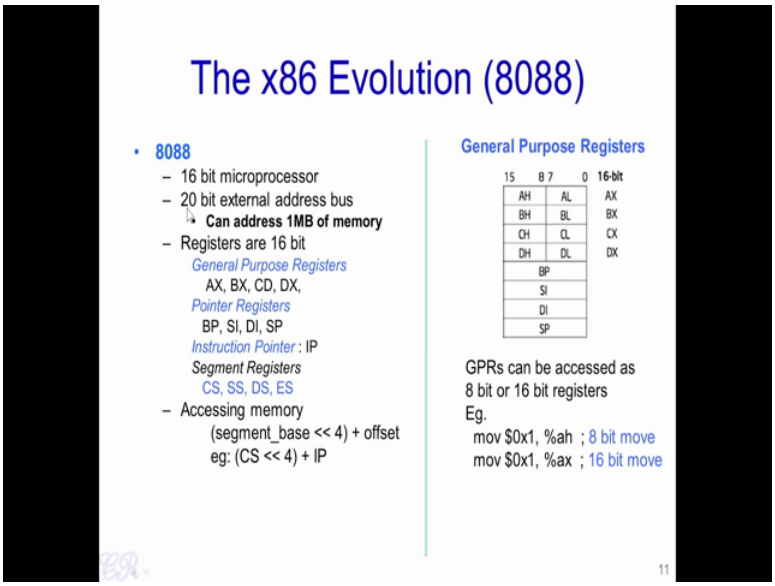
So, this particular slide here shows how a typical PC is organized. So, we may have multiple processors present in the system. So, this could be either 1 processor, 2 processor, 3 or 4 and so on. So, it could also be possible that each of these processors have multiple cores inside them; and each core has 2 or 4 threads. Now all these processors are connected to each other through what is known as the front side Bus. So,

there is always a sharing between all processors with respect to the front side Bus. Now an important device which connects to the front side Bus is known as the North Bridge or the chip set. So, the North Bridge would interface with the memory through what is known as the memory Bus; and it also interfaces with the PCI Bus.

So, on the PCI Bus you could have several devices like the Ethernet controller, USB controller. And the USB controller could have many USB devices, which you see at the front or the back of your desktop. Now as you would notice over here, the USB devices are connected in a tree like structure.

Now the PCI Bus also has a hierarchy type of structure; and each Bus for instance, in this case, PCI Bus 0 which is the closest to the north bridge. And you have in this way a PCI Bus 1, which is connected to the north bridge through the PCI Bus 0. So, the interface between Bus 0 and Bus 1 is through a PCI to PCI Bridge. In addition to this, there is what is known as the south bridge and this is south bridge interfaces with the PCI Bus or you could have a special protocol or a special connection with the north bridge which is known as the DMI Bus. So, in the south bridge, various legacy devices like the PS 2 devices, keyboard, mouse, PC speaker, and so on are connected.

(Refer Slide Time: 21:03)

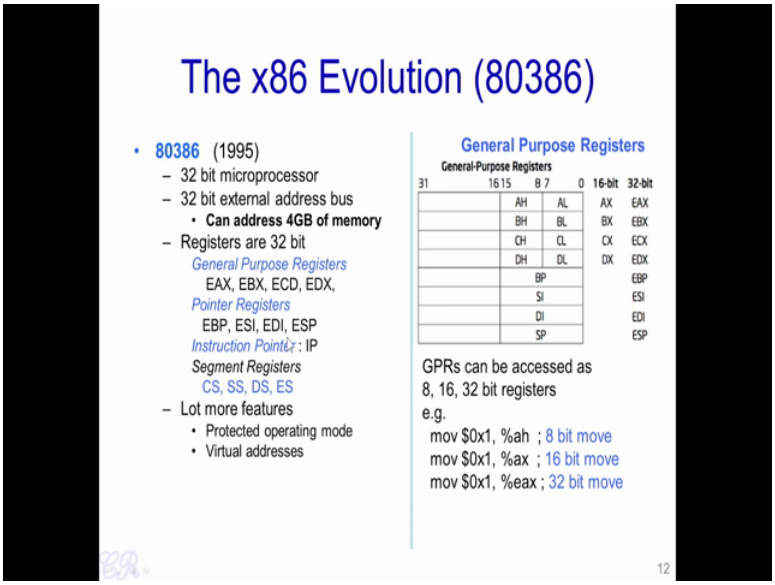


So, Let us start with how x86 or the Intel x86 processor evolved. So, it all started with the 8088 processor or the 8088 processor as it was generally known as. So, this was a 16 bit microprocessor which had a 20 bit external address Bus. And therefore, could address up to 1 MB of memory. So, how did we get this 1 MB is essentially 2^{20}, so that 2^{20} is 1 MB. So, the registers within the 8088 were 16 bit; and it could be divided into various types such as the general purpose registers that is the AX, BX, CX and DX. We had pointer registers which were used to point to string which are stored in memory, so these were the base pointer, stack index, destination index and the stack pointer. So, as we know that the base pointer is used to point to a frame present in the stack, where the stack pointer is used to point to the bottom of the stack.

Then we have the instruction pointer, which points to the instruction that is being executed. And you had several segment registers such as the code segment, stack segment, DS and ES segment registers. Now, in order to load or store an instruction or data in memory what the 8088 processor would do was, it could take the segment base that is one of the segment registers which forms the base, it could left shift it by four and add an offset.

For instance, in order to fetch one instruction from the memory into the processor, the CS register would be used. So, CS register would be the base for the code segment. It would be left shift it by 4. And the IP of the instruction pointer would be added to that. In this way, even though there are the registers used are 16 bit, so each of this registers CS and IP are 16 bit, still by shifting it by 4 bits and adding the IP. It was possible to address 2^{20} memory locations.

(Refer Slide Time: 23:38)



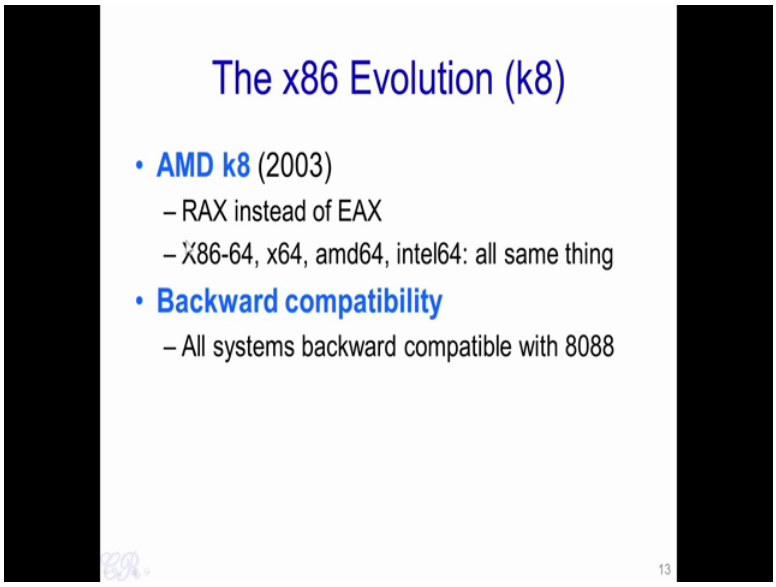
One big step in the Intel systems came when the 80386 was invented in 1995, so these are 32 bit microprocessors. So, this particular processor had a 32 bit external address Bus. So, therefore, it could address 2^{32} memory addresses. So, this was 2^{32} is 4 GB of memory. So, what this means is that a system could have up to 4 GB of RAM present in it. Now the same amount of registers which were present in 8088 are also present here except that now it is extended from 16 bit to 32 bit, and because of this extension the registers are called EAX, EBX, ECX, EDX, EBP, ESI, EDI and so on.

There were also a lot more features like the protected operating mode as well as virtual and segmentation schemes, which were present. So, one thing you would notice is that while all the registers were extended to 32. The segment registers continued to be 16 bit. Now Intel ensured that even though we switched from a 16 bit processor to a 32 bit processor, still backward compatibility with the old 8088 systems was maintained. This meant that any software which was written in an 8088 or 8086 processor would straight away run in an 80386 without much of a problem.

To ensure this, even though the registers were extended from 16 bit to 32 bit still programs could access the registers as if they were 16 bits only available. For example, the AX, BX, CX, DX would access the lower half of the extended registers that is

registers from 0 register with 0 to the bit 15. So, this particular feature in the 80386 processor was taken forward to the 486 Pentium and so on.

(Refer Slide Time: 26:10)



Now more recently in 2003, there was the next big step in the x86 Evolutions. So, this was when the AMD k8 was introduced. So, this k8 moved from a 32 bit processor to a 64 bit processor. As a result, the registers which were known as the extended registers and were of 32 bit, where now extended further to a 64 bits. So, the EAX register which was previously called in the 32 bit platforms like the 80386 was now called the RAX register, which essentially was for 64 bit. So, in this way other registers were also extended to 64 bits. So, in spite of extending from 32 to 64, the companies Intel as well as AMD ensured backward compatibility that is a program written in 8088 would still under certain circumstances will be able to run in the 64 bit platforms as well.

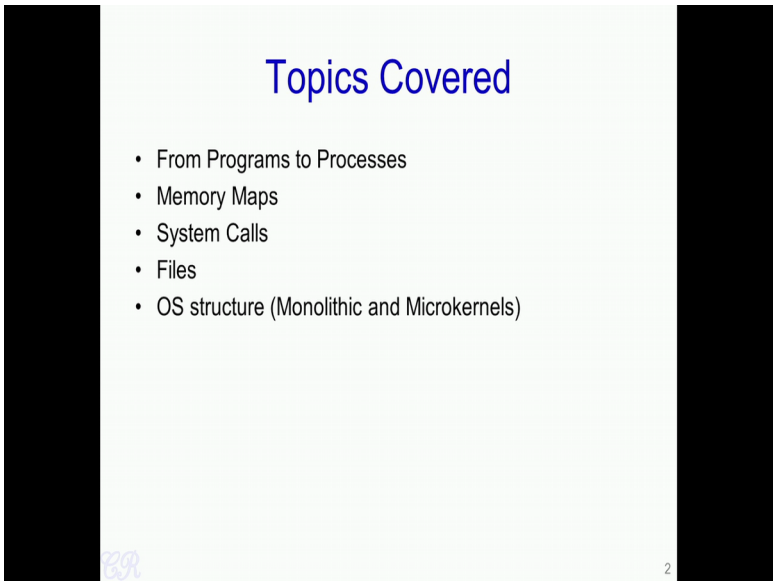
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 01
Lecture - 03
From Programs to Processes

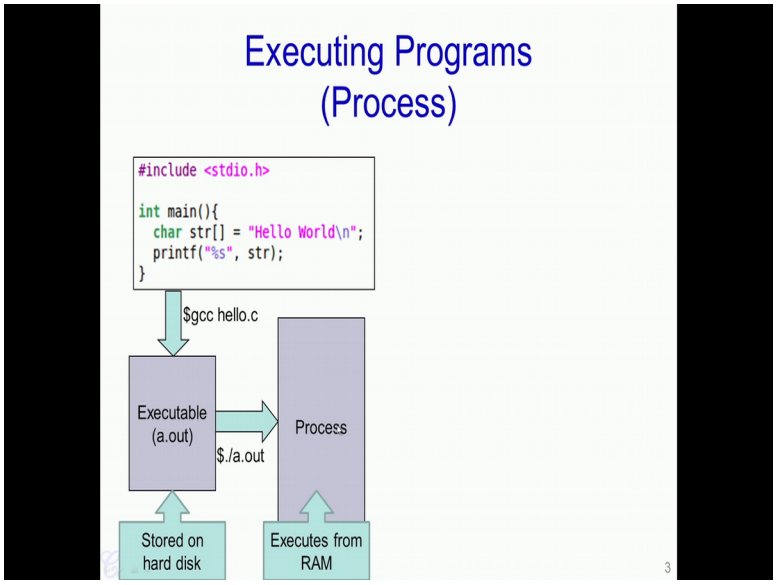
Hello. In today's class we will be having a very brief introduction to an operating concept called Processes.

(Refer Slide Time: 00:23)



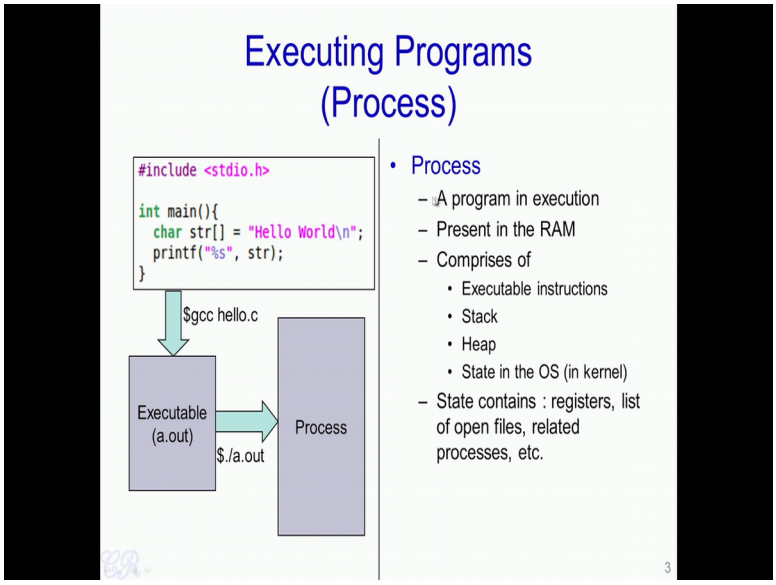
So, the topics which we will cover, is from Programs to Processes, Memory Maps, a System Calls, Files and essentially the structure of the Operating System.

(Refer Slide Time: 00:37)



Consider this particular program written in C. So, this program prints "Hello World" on to the screen. In order to compile and run this program, we first need to use a compiler such as gcc and specify the c code name such as hello.c in this case and what we will get is an executable, in this case it is called a.out. This executable is stored on the hard disk. In order to run this particular program we specify a command like ./a.out and it results in a process being created in the RAM. So, this process is essentially the - a.out program under execution, which is present in the RAM.

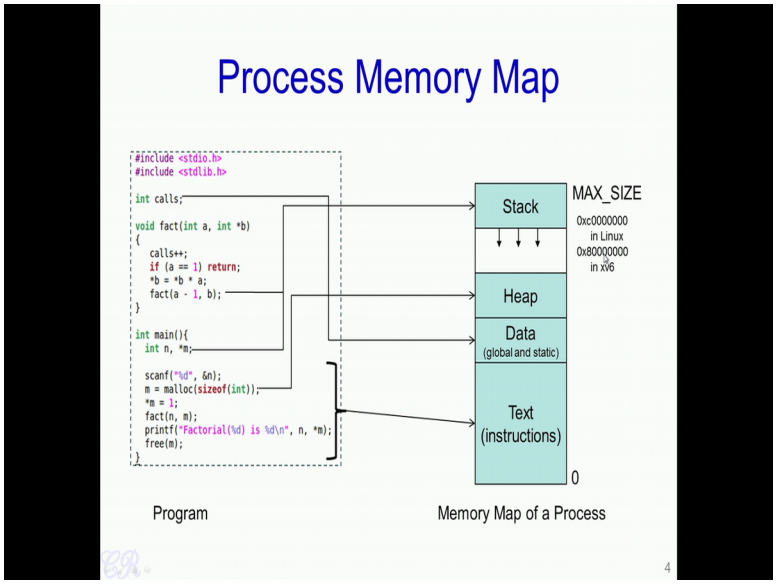
(Refer Slide Time: 01:18)



To define it formally, a process is a program under execution which is executed from RAM and essentially comprises of various sections, such as the Executable instructions, Stack, Heap and also a hidden section known as the State. So, this state is actually maintained by the operating system and contains various things like the registers, list of open files, process, list of related processes etc..

So, in today's class we will look more into detail about what this particular process contains and how it is managed.

(Refer Slide Time: 01:55)



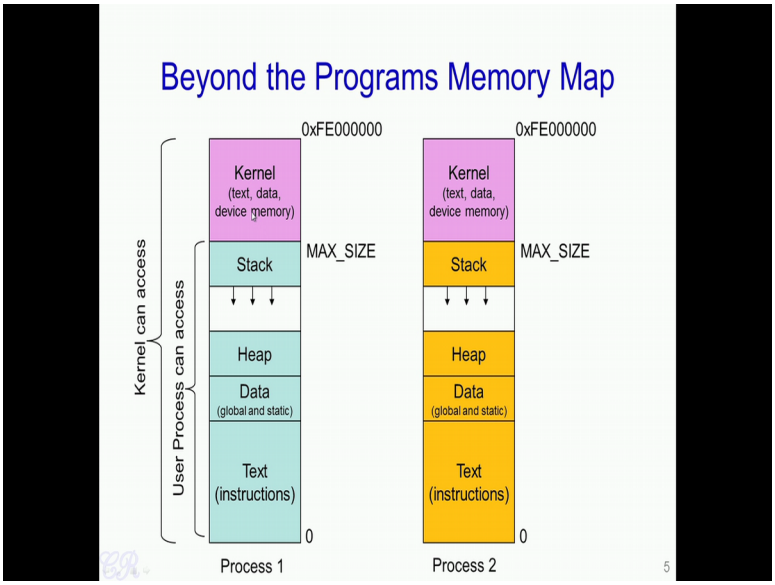
So, Let us take a very simple example. So, this is a program and this is a process that is created when this program is executed. Now the process has various sections for example, it has the Text, Data, Heap and the Stack. Now various parts of this program when executed get mapped into the various sections of the process. For instance, all the instruction such as the instructions involved in the function main will get mapped into the text section of the process. Similarly, other functions such as the fact() function (mentioned in above slide) the instructions involved in this will also get mapped into the text section.

Now the global data and also static data gets mapped into the data section of the process. So, this section is actually divided into two parts where called as initialized and non-initialized sections. Third section is the heap, now any dynamically allocated memory such as m (mentioned in above slide) which is dynamically allocated using malloc gets created in the heap. Now the final section is called the stack, which contains all the local variables such as n and m and also information about function invocation. For example, in this case we have a recursive function which is getting invoked. So, all this information is present in the stack.

Now, the memory map of a process comprising all of the sections has a maximum limit

called the MAX SIZE. So, typically at least in processes which are used in typical operating systems these days, this MAX SIZE is going to be fixed by the OS. For instance in a 32 bit Linux operating system, the MAX SIZE of every process is fixed at 0xc0000000. In the xv6 operating system which we are looking at for this course, the MAX SIZE of a process is fixed at the address 0x80000000.

(Refer Slide Time: 04:07)



So, what we had seen is that if every process, a program that is a program under execution gets mapped into an area which starts at 0 and ends at MAX SIZE. So, what is present beyond this MAX SIZE of the process? So, typically the Kernel or the operating system gets mapped to the memory region from the MAX SIZE to the maximum limit. The entire thing like Text i.e the instructions of the operating system, operating system data, the OS heap and also device memory gets mapped into this upper region of the OS.

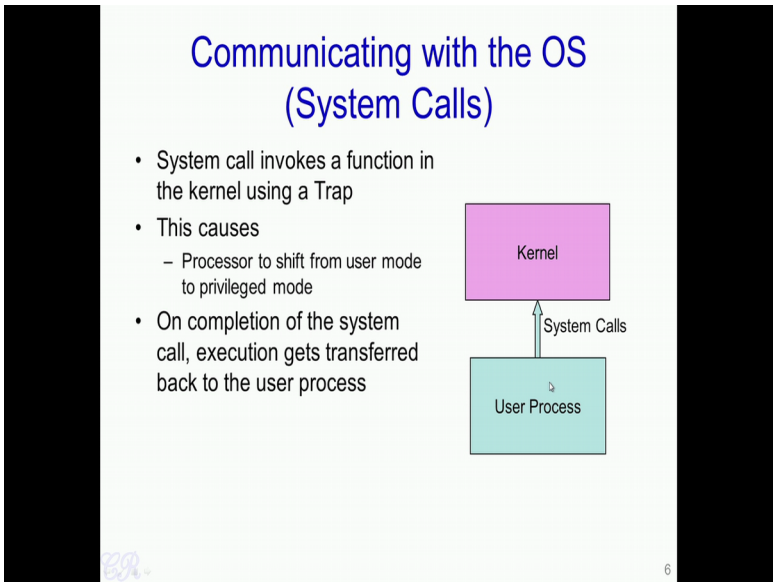
So, typically any user program could access any part of this lower region i.e the green region (mentioned in above slide). So, there would not be any problem to actually read data from any of these user space regions or even write data to parts of these user space regions. But, however, the process cannot access any data present in the Kernel memory that is beyond the MAX SIZE limit. However, the Kernel or the operating system which is executing from this upper region can access data from any part of the region that is it

could execute or access data from in this kernel region as well as in the user space region.

Now what happens when we actually have multiple processes running in the same system? So, each process would have its own memory map, we having its own instructions, data, heap and stack, and also the kernel component is also present beyond the MAX SIZE. So, what you see is that every process in the system would have the kernel starting at MAX SIZE and extending beyond. Only the lower parts and this kernel part is going to be same for every process that is executing in the system. Below this MAX SIZE is going to vary from one process to another.

So, what does this mean? So, what it means is that when you execute one process and then executing another process, the regions above the MAX SIZE is going to be similar, while the regions below the MAX SIZE is going to change from one process to another. So, we mention that user programs will not be able to access any data in the kernel space. So, in that case how does the user program actually invoke the operating system?

(Refer Slide Time: 06:40)

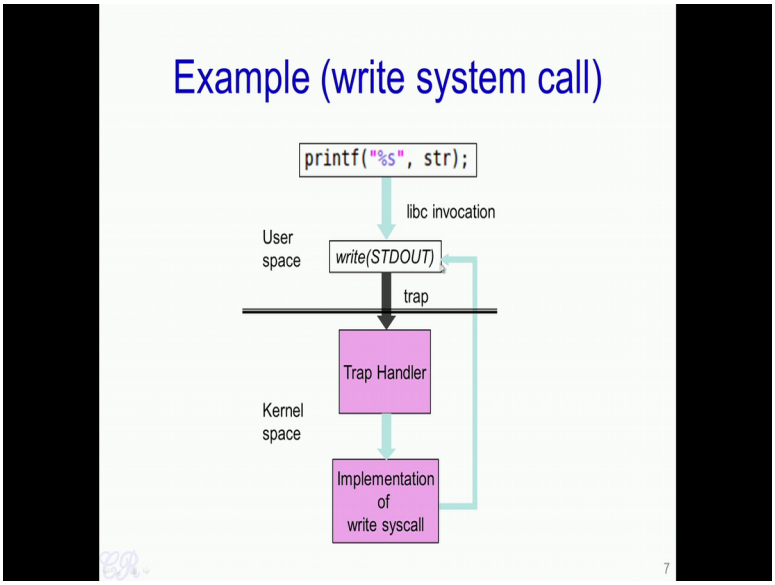


So, there are special invocation functions which the operating system support these are known as System Calls. So a System Calls are a set of functions which the OS support

and a User Process could invoke any of the system calls to get information or to access hardware for other resources within the Kernel.

So, what happens when a system call is invoked; is that a process which is generally running in a user mode gets shifted to something known as a kernel mode or a privilege mode which will allow the kernel or the operating system to actually execute. When the system call completes execution then the user process will resume its execution from where it actually stopped.

(Refer Slide Time: 07:25)

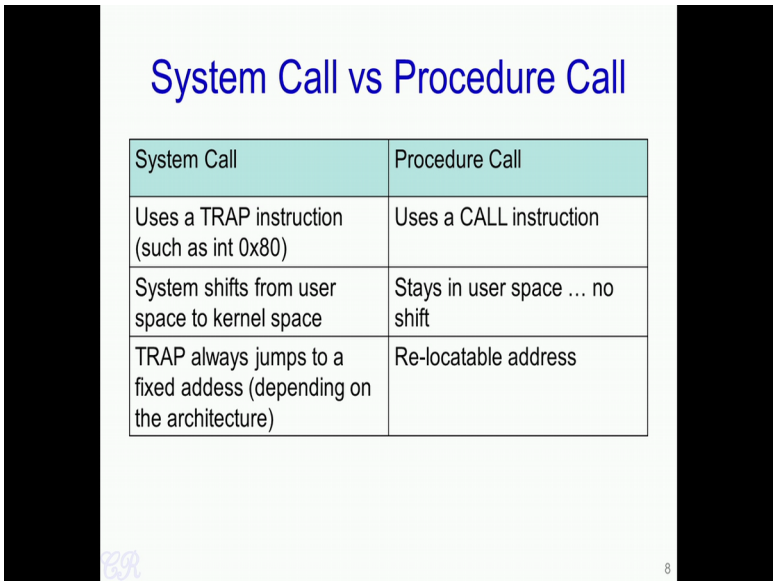


So Let us take an example of the printf statement. So, printf in fact is a library call. So, it is present in this libc and it results in particular function in the User space known as write() to be invoked and printf() function will then invoke a system call called write, with the parameter call STDOUT. So, STDOUT is a special parameter which essentially tells the operating system that this string provided by printf should be displayed on to the standard output that is the screen.

So, the write is the system call which causes a trap to be triggered, and this trap will result in something known as a Trap Handler in the Kernel space to be executed and the Trap Handler would then invoke a function which will correspond to the write system

call. So, this write system call will then be responsible for actually printing the string provided by str on to the screen. After the write system call completes, then the execution is transferred back to the user space and the process continues to execute.

(Refer Slide Time: 08:40)

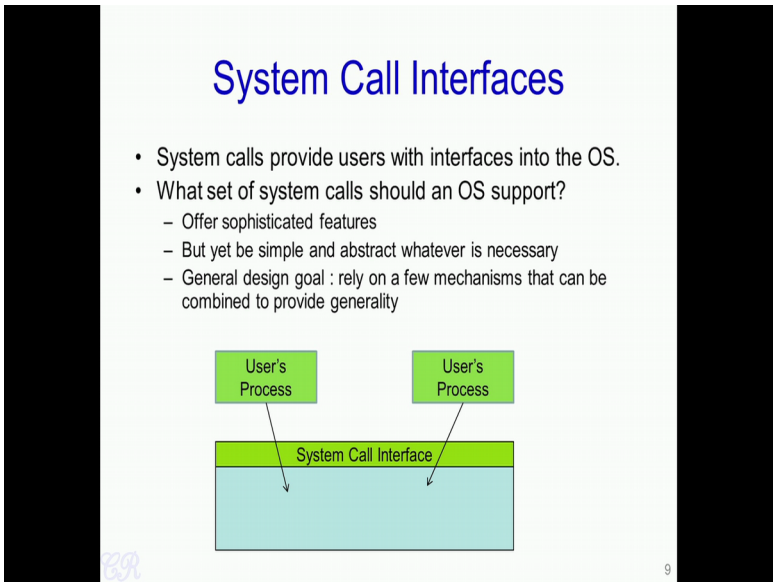


What is the difference between a System Call verses a Standard Function Call or Procedure Call. So, one important difference is that, when we want to invoke a function in a program or in a process we use an instruction such as a CALL instruction this is a standard x68 assembly instruction, and this will result in the function getting called and after that function gets invoked it returns back to the calling function.

In order to invoke a system call however, we use a TRAP instruction such as the int 0x80. So, int here stands for interrupt or software interrupt and it results in the system shifting from the user space or the user space mode of operation to the kernel space. So, the trap instruction causes the kernel to be invoked and it causes instructions in the kernel to then be executed. However, when we use the standard function call or the procedure call using the CALL instruction there is no change or shift from user space to kernel space and so on. So, the execution continues to remain in the user space as it was before.

Another very settle difference between the system call and a standard procedure call or a function call is that the destination address or the destination function which is invoked can be at a relocatable address. So, it could change every time the program is compiled and so on. However with the system call when a trap instruction is used the hardware actually or the processor actually decides where the next instruction in the kernel space should get invoked. So, this is going to be fixed irrespective of what program is running, what operating system is running, and so on.

(Refer Slide Time: 10:30)

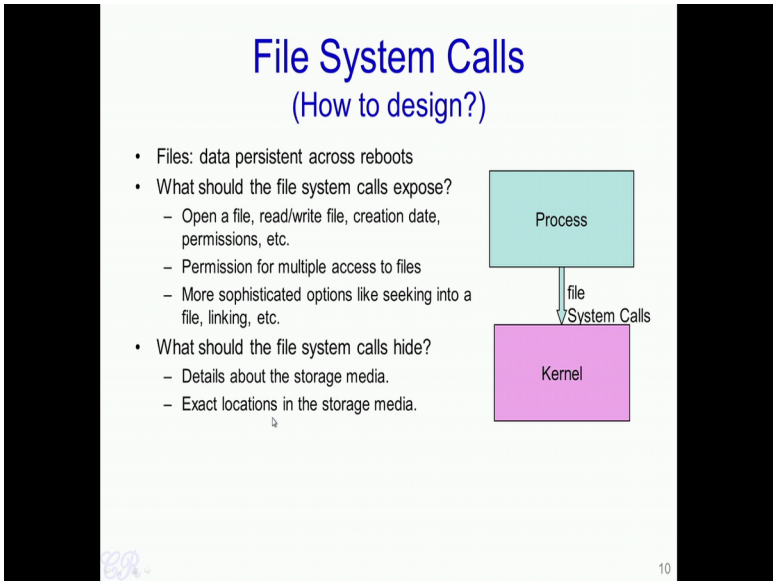


So, one crucial aspect when actually designing operating systems is what system call should the operating system supports? We had seen that the only way a user process could invoke a particular functionality in the operating system is through the system call interface. So, the question now comes that if a person is actually designing an operating system, so what are the interface that the system call should support.

So, one obvious requirement is that the system call interface should have several sophisticated features so that a user process could actually very easily be able to interface or invoke several important functionality in the operating system. However a different approach is to have a very simple system call interface and abstract whatever is necessary from the operating system. So, we will see in the next slide, a particular

example in this.

(Refer Slide Time: 11:29)

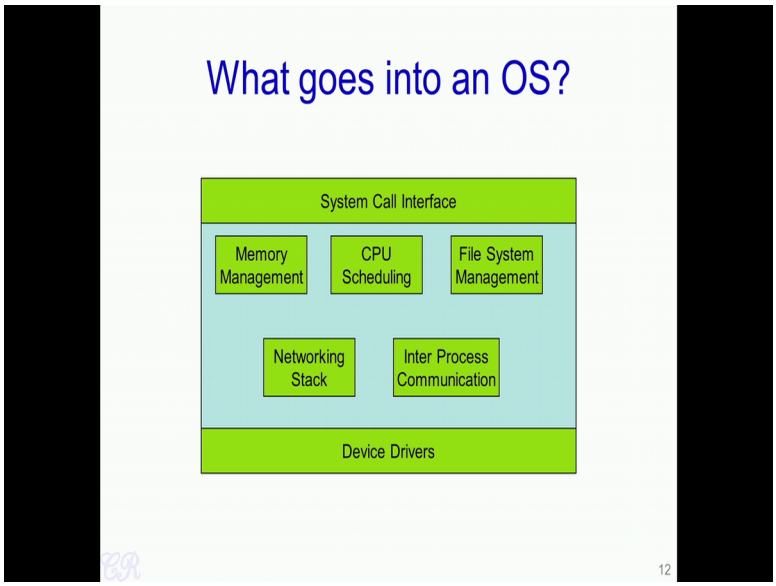


Let us take an example of system calls that an OS supports for accessing files. So, as we know files are a data which is persistent across reboots, so these are data which is stored in the hard disk and could be read, written or accessed using function such as fopen, close, read, write and so on. Every time we do a file open, it would require that the hard disk be accessed, so a process would need to invoke a system call into the kernel and the kernel should actually take care of accessing the hard disk or a hard disk buffers or any other storage medium to open the file and return back a pointer to the process.

Now the question comes is how does a operating system designer decide what system call should be provided or supported in order to access files? So, some of the obvious things are like there should be system calls to open a file, read or write to a file, there should be system calls to actually modify the creation date, set permissions and so on. The operating system could also support more complicated or more sophisticated operations such as being able to seek into a particular offset within a file, be able to link between files and so on. So, these are the essential requirements that a system call for handling files should support.

On the other hand, operating system should be able to hide some details about the file. For instance, details which should not be supported by system calls is like things such as like details about the storage media where the file is stored, for instance whether the file is stored on a USB drive or a Hard disk or a CD-ROM. This is actually abstracted out by the operating system, and the user process would not be easily able to know where the file is stored. Another aspect which is abstracted out by the operating system is the exact locations in the storage media.

(Refer Slide Time: 13:40)

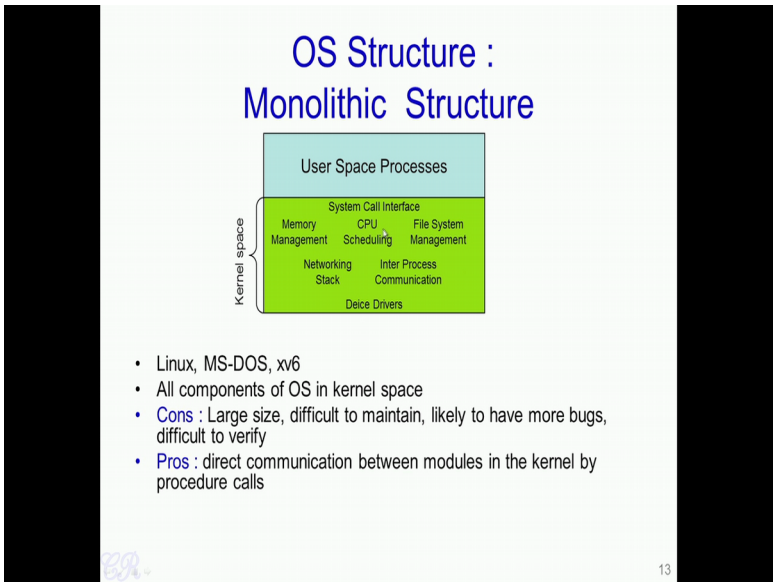


We will now look at how a typical Operating System structure looks like. So, the Operating System, suppose we consider this as a big green block have a several modules built internally. For example, it would have a memory management module which manages all the memory in the system, it would have a CPU scheduling block that is also the file system management module which will control how the file system such as the those present in hard disk or a CD-ROMs are managed. So, you have a networking stack which manages the TCP/IP network and you have something known as the inter process communication module which would take care of processes communicating with each other.

So, two important things which have not been mentioned as yet is the System Call

Interface, which allows user processes to actually access features or functionalities within each of these modules. Another aspect is the Device Drivers which would take care of communicating with the hardware devices and other hardware resources within the system. So, this essentially is all the different modules that an operating system supports.

(Refer Slide Time: 14:52)

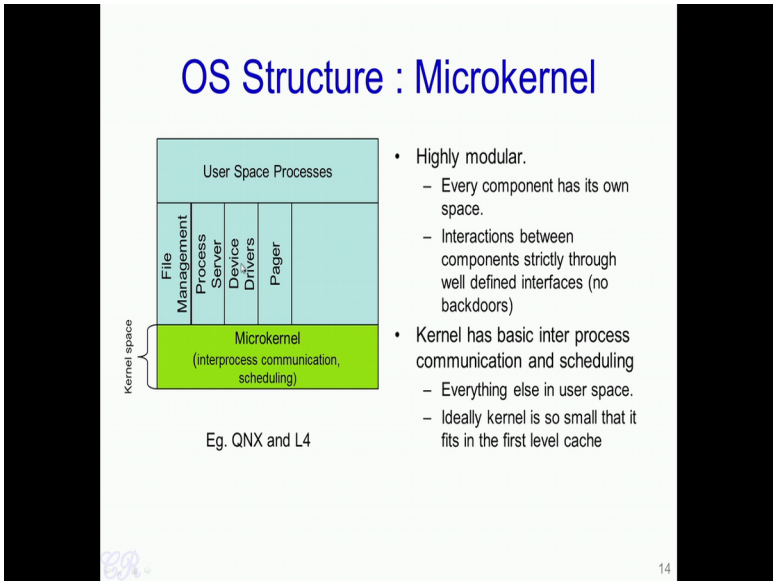


So, In a Monolithic Structure of an operating system, all these various modules in the OS are present in a single addressable kernel space, so what this means is that this is just one large chunk of code and all of them are you could think of as one large program where all these modules are present in. So Therefore, calling any function from the memory management to say the networking stack would just mean a simple function call. Similarly from the networking stack to the device driver would be another function call.

This is essentially the advantage of having such a monolithic structure, is that you could there is a direct communication between one module and another. On the other hand the Kernel space becomes very large, and therefore, difficult to maintain and is likely to have more bugs. So, typical operating system such as Linux and xv6 and MS-DOS uses a monolithic structure. So, to take an example the Linux operating system or the Linux Kernel has around 10 million lines of code. So, all these 10 million lines of code

comprises of the entire kernel Linux Kernel which is actually present in this area.

(Refer Slide Time: 16:14)



Another common structure of the operating system is known as the Microkernel structure, where the kernel is actually highly modular and every component in the kernel has its own addressable space. So, it is like having each of these as independent processes and you have a very small microkernel which actually runs in the kernel space, which is in charge of managing communication between each of these processes, and also communication between user process and the operating system processes.

So, the advantage here is that this microkernel is extremely small. So, ideally it is small enough to actually fit into the L1 cache of the system itself, so typically it would be quite fast. However, the drawback is that you now cannot have direct calls from say the file management to a device driver or rather like unlike the monolithic kernel where you could make direct function invocation from a file management module to a device driver function. Here every invocation of that form should be through a communication channel known as an IPC or Inter Process Communication channel.

With this we would actually end today's lecture, and from the next lecture we will actually look more about CPU sharing.

Thank you.

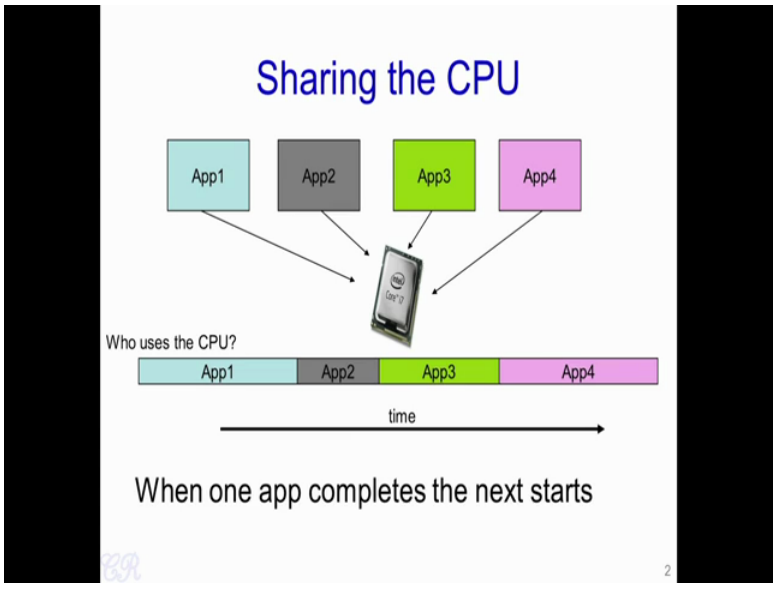
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 01
Lecture - 04
Sharing the CPU

Hello. In the introductory video, we had seen that one of the most important applications of an operating system is for resource management. Essentially, since the system that the OS runs on has limited amount of hardware resources, so the operating system is in charge of utilizing these resources in an efficient way.

Among all the resources that are present in the system probably the most critical resource is the CPU. So, most systems have a very few number of CPUs; earlier systems in the especially the earlier systems in the 1990s and the early 2000s had a single processor, while systems these days are equipped with 4, 8 or 16 CPUs. So, it is important that the operating system ensures that these CPUs present in the system are adequately utilized. So, we will see how these CPUs are managed by the OS.

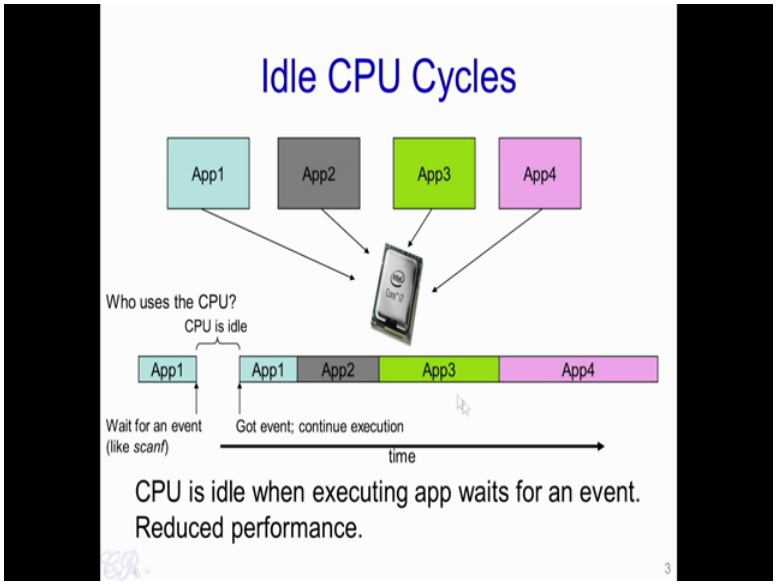
(Refer Slide Time: 01:32)



In systems these days, we have several applications that are running contiguously. So, we had seen that we could have something like Office application, Internet explorer, a Skype application and a Power point application all of them need to share the single CPU. So, how is this possible, essentially how does the operating system decide who should run or rather which application should run on the CPU?

One very simple thing that the operating system can do is to put one application onto the CPU, and let it run till it completes. For instance, a very primitive operating system such as the MS DOS operating system would start an application and make it execute in the CPU until it completes. When that application completes then the next application that is App 2 would execute in the CPU till it completes, then application 3 would start and execute in the CPU till it ends and then application 4 would start. While, this is a very easy way to manage the CPU time, it is not very efficient and we will see why.

(Refer Slide Time: 03:10)

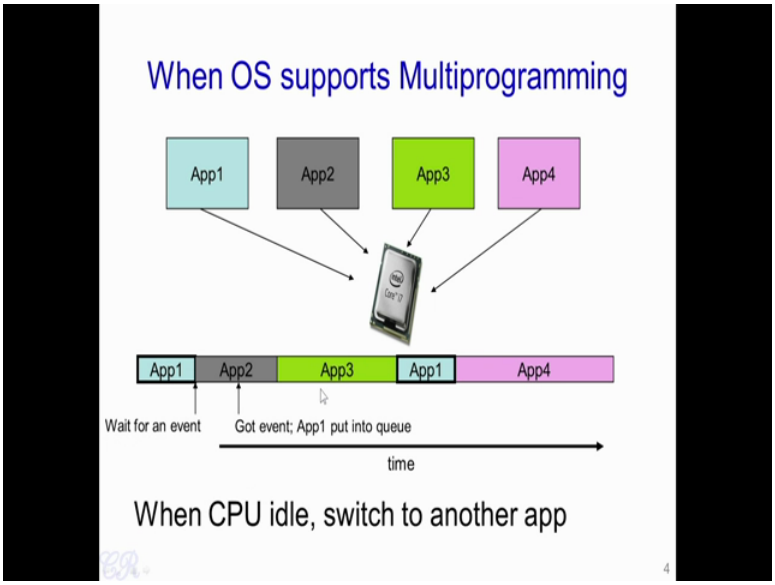


So let us say application 1 is executing on the CPU and after sometime it waits for a particular operation to be done by the user. For instance, the application1 is waiting for an event like a scanf. Essentially it is waiting for the user to input something to the keyboard. So, from this time onwards until the user actually presses a key, the CPU does nothing but is waiting idly. So essentially it is wasting time, so these are the idle cycle

times of the CPU. Only when the event is obtained, will the application 1 continue to execute.

Thus, we see that even though the scheme of executing 1 application after the other completes is very simple, yet it does not utilize CPU time adequately or efficiently because every time that the application requires an event, the CPU would be idle until that event actually occurs. And this as we see would cause a reduced performance with respect to the utilization of the CPU.

(Refer Slide Time: 04:46)

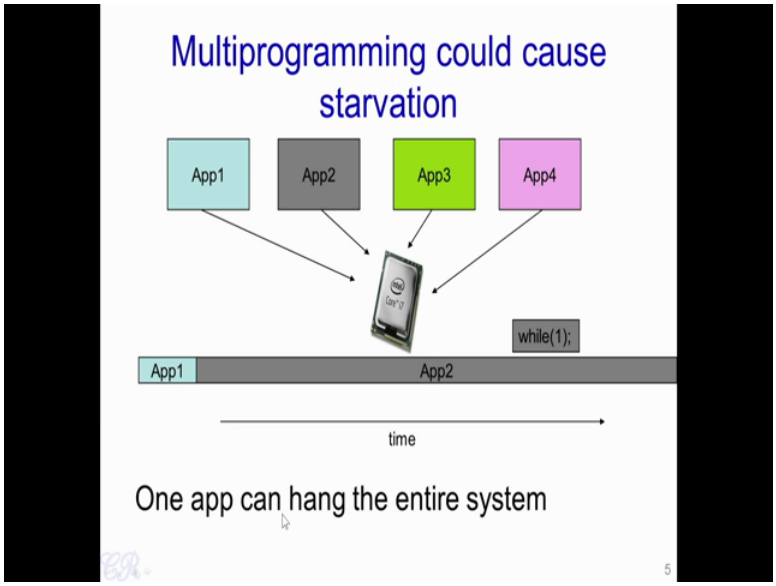


So a better way to do is something known as Multiprogramming. So, in an operating system that supports multiprogramming, the application 1 will continue to execute until it requires an external event. So, when a function like scanf gets executed in App 1 that application will become blocked. Essentially it will be preempted from the processor and another application in this case app 2 will execute in the CPU. So after a while, when the user inputs something through the keyboard, the event is obtained and as a result of this application 1 is put into a queue.

At a later point in time, application 1 will be again given the CPU and will execute from where it has stopped. Essentially it had stopped during the processing of the scanf, and it

will continue to execute from where it had stopped. Essentially now it has got the input character which the user has pressed on the keyboard and it will continue to execute from this point onwards. Therefore, we have seen that we are preventing the CPU from being idle. Essentially by executing an other application on the CPU, we have preventing the CPU from idly running and therefore, increasing performance. However, there is a problem with this also.

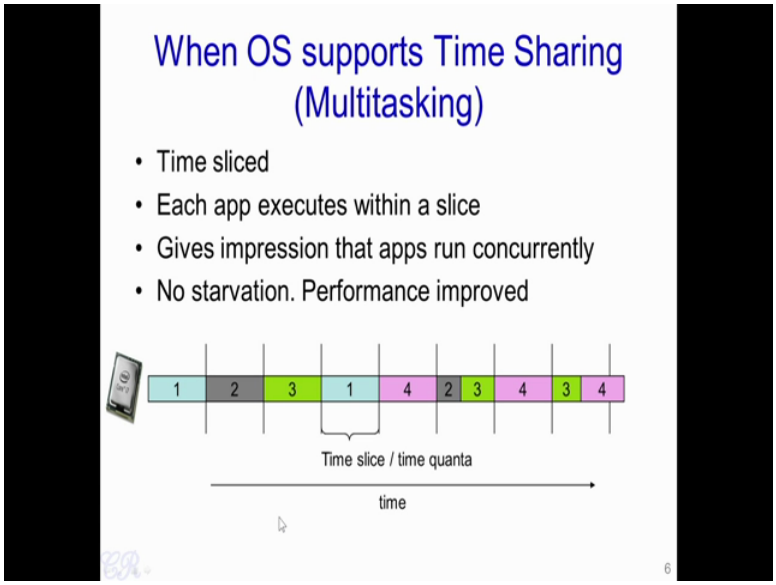
(Refer Slide Time: 06:41)



Now, consider this particular scenario where application 1 runs for sometimes and then gets blocked. However, application 2 has something like this (mentioned in slide) present in its code; essentially this is an infinite loop. So, while(1) would continue to execute infinitely. So, as a result of this app 2 will never stop, it will keep executing and holding onto the CPU.

So, due to this other application such as app 3, app 4 and as well as app 1, when once it has obtained the event will never be able to run. The only way to terminate this particular thing would be to forcefully stop application 2 or reboot the system. In earlier operating systems like MS DOS, this use to happen quite frequently. As a result of this, it was often require that the CPU be restarted because of one application that is hanging the entire system.

(Refer Slide Time: 07:55)



More recent operating systems support what is known as Multitasking or Time Sharing. Essentially, in this case, the CPU time is split into slices or so each slice is known as a time slice or a time quanta. In each time slice, a process would run; the process will continue to execute until one of two things happen.

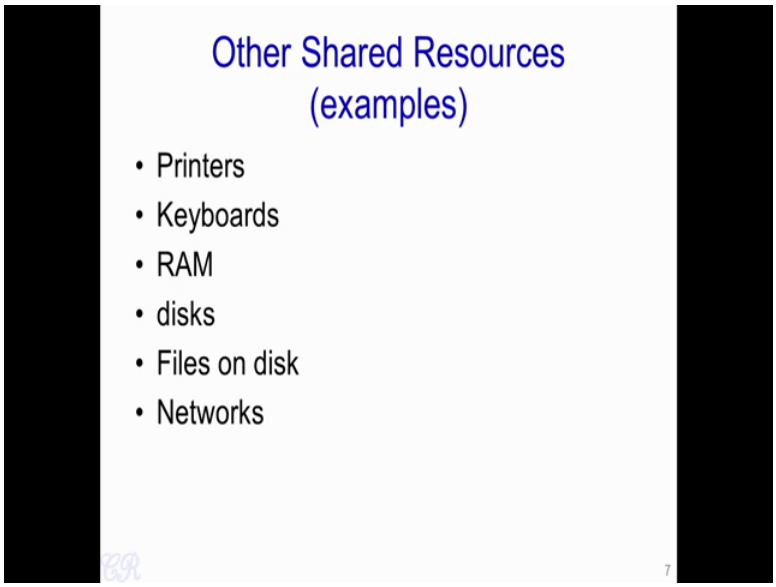
First, it will execute until its time slice completes; in which case another process is given the CPU. So, for instance, over here process 1 executes until its time slice completes, and then process 2 is given the CPU and the process 2 would execute. Then when process 2 completes its time slice, process 3 will be given the CPU and will continue to execute. Now, in this way all processes are sharing the CPU. After sometime the processes could continue to execute from where it had stopped. For instance process 1 had executed till its time slice and after sometime based on some decisions by the operating system process 1 would be given the CPU again to execute. At this point in time process 1 will continue to execute from where it had stopped.

From a user's prospective, this delay as a result of multitasking that is as a result of other processes being executed or rather said another way, the delay or the intermediate execution of a single process is hardly noticeable because the time slice is very smaller.

The other way that a process stops executing is when it requires to wait for an event as we have seen in the previous slide or it terminates.

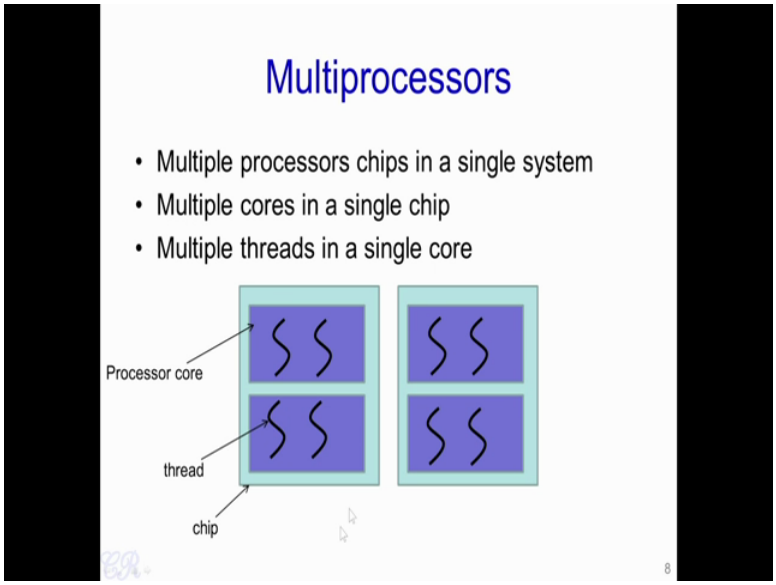
In either way, it results in another process being offer the CPU and the new process will execute. So, the advantage of multitasking is that from a user's perspective, it gives the impression that all the applications are running concurrently, there is no starvation. Also over all from a system perspective the overall performance is improved.

(Refer Slide Time: 10:52)



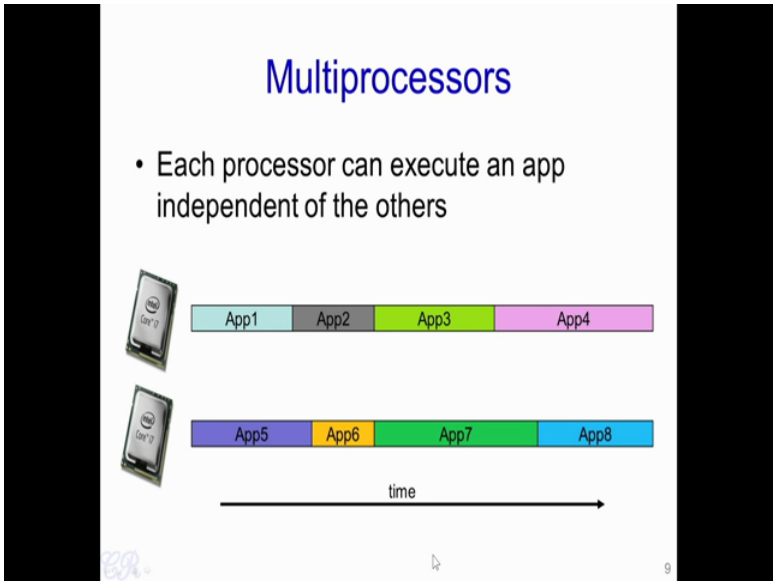
So, the CPU was one example of a resource which the operating system managed. So, essentially the operating system decided which of the several processes would be given a chance to execute in the CPU. So, along with the CPU, the system would have several other resources; and the operating system would need to share these resources among the various users i.e among the various applications. For instance, the printers which are connected to the system, the Keyboards, RAM, disks, the Files on the disk as well as networks. All these are resources in the system and the operating system should manage these resources and ensure that all applications have a fair usage of these various resources.

(Refer Slide Time: 11:54)



So, what would happen in the case of Multiprocessors? Essentially, where the system has more than a single processor, such a system would look something like this. Essentially, there would be several chips or CPU chips and each chip would have several cores. For instance, in this case there are two chips, CPU chips; and each CPU has two cores; and further it is possible that each core runs multiple threads. So, this is achieved by a technique known as Symmetrical multithreading. In the Intel nomenclature this is known as the Hyper threading. So in this particular system, which has two chips, a total of two cores per chip and a total of two symmetrical threads per core, the CPU has to manage all these resources.

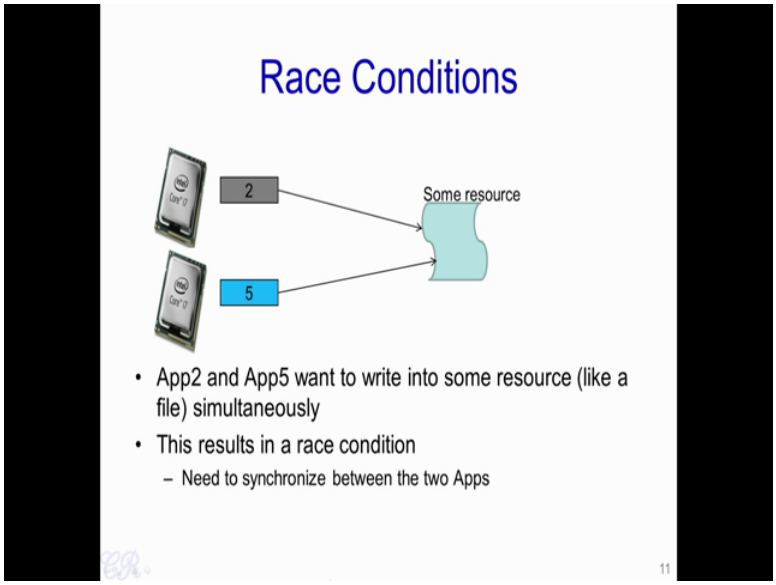
(Refer Slide Time: 13:07)



So essentially in such a case, the CPU has to ensure that all computing environments in the system are adequately used. So in such a case, we would obtain something as parallelism in the sense that suppose we have 2 processors over here and let us assume that each processor has a single core in them. So, it would mean that the operating system could then schedule two processors or rather two applications to execute simultaneously on the processors. So, one application will execute on one processor, while the other application executes on the second processor. In addition to this, time slicing is possible on each processor i.e each processors time could be split into time quanta or time slices as we have seen before and shared among the various applications.

So in this case, for example, each processor has a time slice which completes periodically; and at start of each new time slice, the operating system running on the processor would ensure that application gets scheduled to execute on that CPU. Similarly, for the second processor, there is also time slices and at the end of a time slice the application being executing will be preempted from the processor and the OS will select another application to execute. So, the operating system should be designed in such a way to ensure that applications unless program to do so, do not execute simultaneously at the same instant in both the processors.

(Refer Slide Time: 15:13)



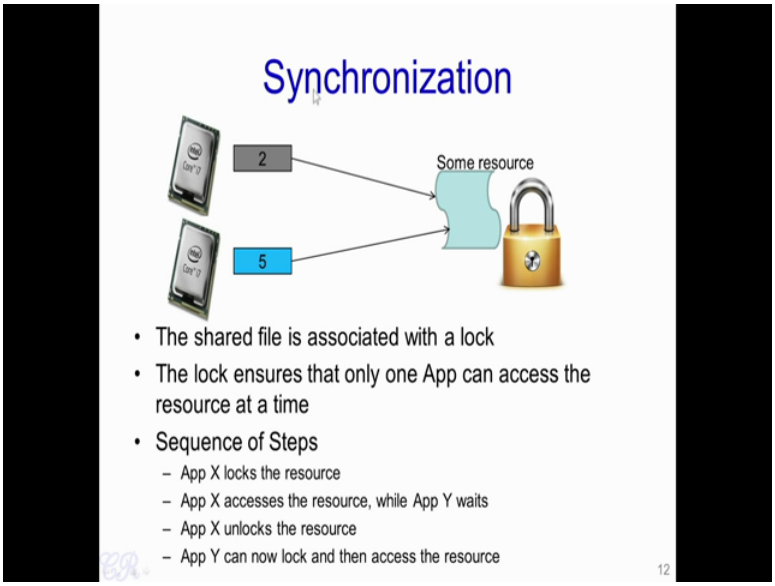
Now one huge issue with a multitasking environment, where time of each processor is split into quanta or time slices allowing different processors or different applications to run concurrently. And also when there are multiple processors present in the system, allowing two or more processors or applications to execute in parallel. So, the issue what arises over here is when two processors or two applications simultaneously request to access some resource.

For instance, let us say this resource is a particular file or it could even a device like a Printer, and we have two applications - application 2 and application 5 want to write or print something into this resource. So, both application 2 and application 5 want to use this particular resource. So, this results in what is known as a race condition; and it results in quite a huge issue when we are studying operating systems or when we are actually looking deep into operating systems.

So essentially, in order to avoid such an issue, the operating systems needs to synchronize between the two applications. It should ensure that were when one process request for a particular resource like a file stored in the hard disk or the printer, an other application app 5 will not be given permission to write or access that particular resource. Only when application 2 complete using its resource, will application 5 be allowed to use

that resource; said another way the operating system will ensure that there is a serialized access to this resource. It will ensure that not more than one application is using this resource at a particular instant of time.

(Refer Slide Time: 17:46)

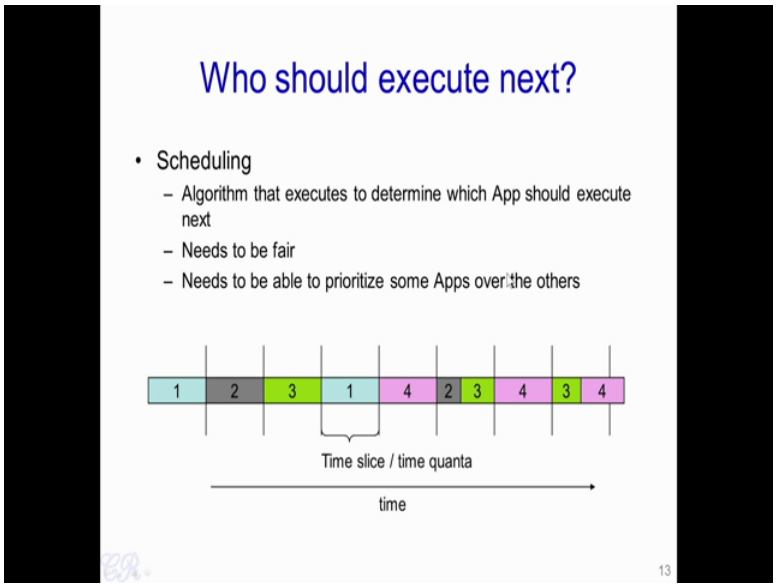


Essentially, operating systems solve these problems by using a technique known as Synchronization. Now, synchronization you could think of is kind of a lock, which is associated with a resource. So, when a application wants to access that resource then it should first get the lock; essentially it should first acquire the lock. So, for instance, we have application X which wants to use the resource; in such a case application X should lock the resource. So, locking the resource would mean that no other application such as App Y or 5 in this case would be able to use this resource.

Now after application 2 completes using the resource, it will unlock the resource. And during that time, if we have a second application which has also requested for the resource, the second application would have to wait. So, when application 2 completes, it will unlock the resource; and this is a signal to application 5 that it can then use the resource. In order that application 5 uses the resource, it will first lock the resource ensuring that no other application can then use a resource; and at the end of its usage, it

will unlock the resource. Unlocking will allow other applications to use this shared resource.

(Refer Slide Time: 19:27)



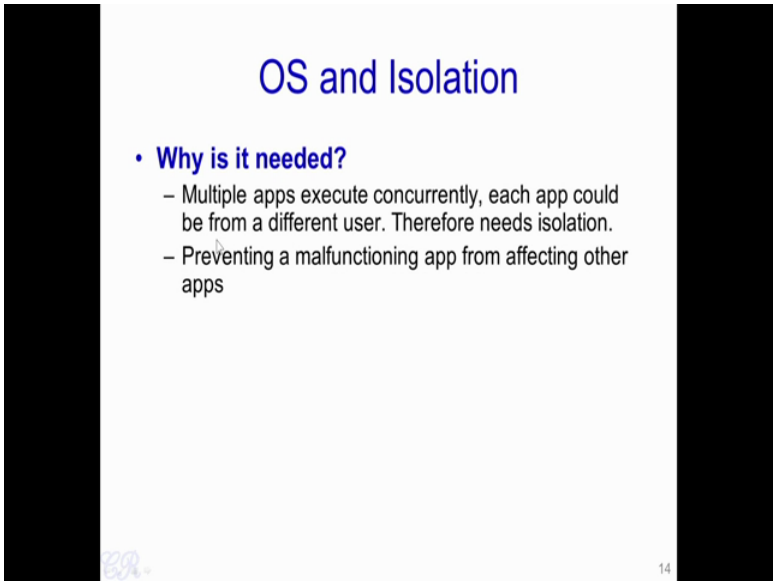
Now, one decision that the operating system needs to make especially in a multi-tasking or multi-programmed based system is the decision about which application should run next. So, as we have seen in a multitasking environment application 1 would execute till its time slice and the operating system then decides which process or which application should run next. So, this decision is made by an entity within the operating system call the Scheduler.

Essentially, the scheduler would be designed in such a way that it needs to be fair which means that it should be design in such way that every process or every application running on that system should get a fair share of the CPU. Also in some systems especially, the scheduler should be designed in such way so as to prioritize some applications over the others. And what we mean by this is that we could have some applications which are far more important than other applications and these applications need to be prioritized. In other words, these high priority applications should be given more CPU time to execute as well as it should ensure that these high priority applications do not wait too long to execute in the CPU.

To take an example, let us say we have three applications - application 1, 3 and 4, which are low priority applications. For instance, it could be like executing a compiler or doing some scientific operations like computing the primality of a number and so on. So, these it could consider as a lower priority application. While process 2 let us assume is a high priority application, in the sense that it could control some parameter in say a robot or it would acquire some information about the environment such as the temperature or humidity of the state of a particular value.

Now, since this application is of a higher priority, therefore the operating system should ensure that this application 2 should be given more CPU time to execute as well as this application should not be waiting too long before it gets the CPU time. So, this entire decisions based on the number of processors as well as the number of CPU cores and the number of threads in each core, and how processor should be executed in which CPU and which thread in the CPU is decided by a scheduler, which executes in the operating system.

(Refer Slide Time: 23:00)



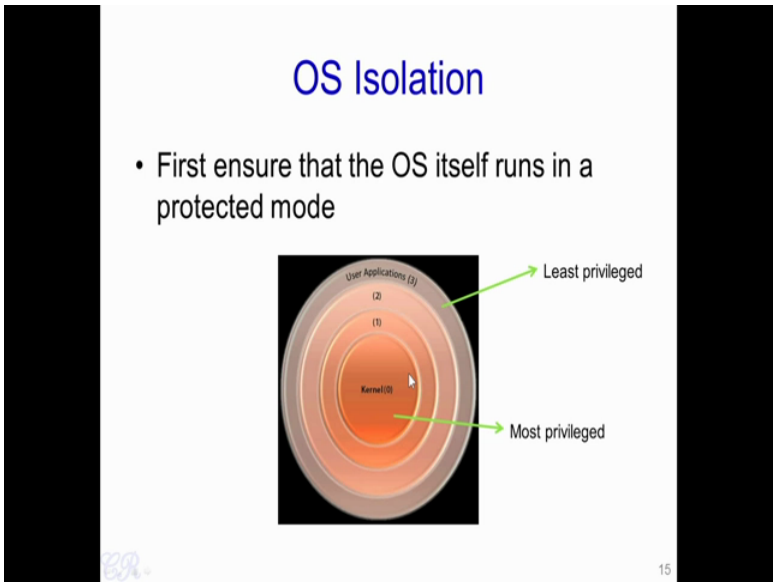
And other very important requirement when we speak about operating system is the requirement for Isolation. Essentially, this arises from the fact that we would have multiple applications that execute either concurrently or in a time sliced manner in the

system. Now these applications could be from different users. Also it could happen that some of these applications are malicious i.e these applications may be a virus or a Trojan which has manage to find its way into the system and is manage to execute in the CPU. Therefore, isolation is required to protect one application from another and thereby the data in one application cannot be visible from another application.

And other requirement for isolation is with respect to the kernel i.e with the operating system. So, the systems are designed in such a way that processors or applications that execute in the system do not directly access various resources.

For instance an application running in the system will not be given direct access to a device such as the printer. If the application wants to use the printer, it needs to go through the operating system i.e it would need to make a system call which in turn would trigger the operating system to execute device drivers with respect to the printers. The application would then send the document or the text file, which needs to print to the operating system, which then transfers it to the printer.

(Refer Slide Time: 25:08)



So, in order to achieve this, most processors and most operating systems have this ring like structure. Essentially, in the Intel platforms i.e the Intel processors, there are 4 rings -

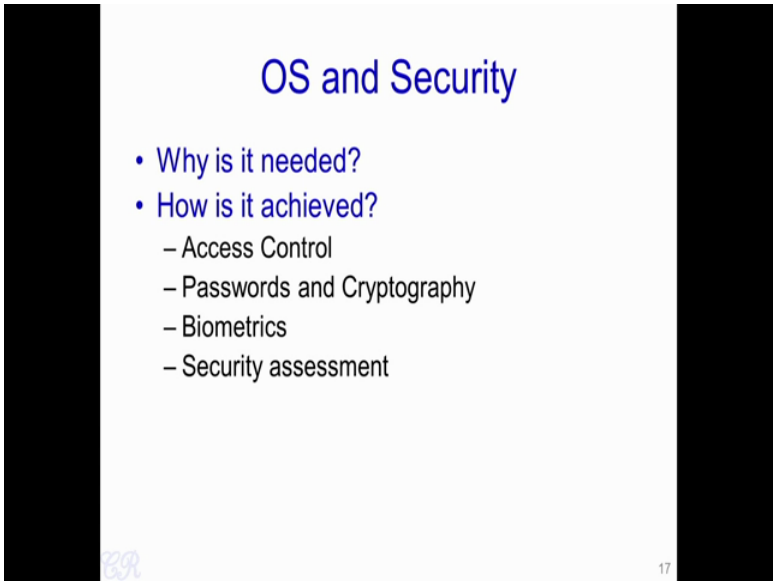
ring 0, ring 1, ring 2, and ring 3. Now, ring 0 is where the operating system executes.So, this is the most privileged mode of operating. So in ring 0, the operating system or the kernel which executes over here can do quite a few things like a manage the various resources, directly communicate with various devices and also could have control about the various applications that are executing. So in modern operating systems like Linux, which run on Intel processors, the third ring that is ring 3 is used by the user applications. So, this is the least privileged ring and it ensures that applications that are executed in this ring do not have access to the kernel i.e an application running in this ring under normal circumstances will not be able to view or determine anything about the operating system.

So, whenever you run a application on your system such as let us say a Web browser or a Office application, so the application would be started as a user application in ring 3, and in order to use a resource such as the network or to display something on the monitor, the these applications would need to invoke the kernel through the system call. Therefore, the kernel is isolated from all the user applications.

In addition to this, the operating systems as well as the processors are designed in such a way that each application that runs in the users space is isolated from each other. What this means is that if we have say of a Web browser running here and also an Office application. The entire system of the operating system along with the processor will ensure that the web browser has no information about the other application that it is it does not know what the Office application is doing. And the Office application will not know what the Web browser is currently executing. For instance, the office application will not know what web pages are currently being browsed.

So, this you could see is useful to isolate different users as well. So, if you have two different users who are sharing the system, so user 1 will not know what the user 2 is executing unless he makes use of certain kernel functions. And the only way to use those kernel functions is through system calls or in some certain cases like Linux, there are certain functionalities which are displayed about the kernel. So, the user will then be able to use those functionalities of the kernel to determine about the other process.

(Refer Slide Time: 29:09)



Another important aspect when are talking about operating systems is the need for Security. Essentially, security features are added to operating systems to ensure that only authorized users are able to use that system. So, if for instance, I do not have a user account in a particular server, then the operating system will ensure that I will not be able to login to that particular server, and I will not be able to run any application in that server. So, how is security achieved in the system is by several features that operating systems support. For instance by having a techniques know as access control, which will ensure that some files that are created by a particular user is not visible to another user.

For instance, suppose I login to a system as a guest i.e through a guest account. So, the operating system will then know about this and determine what files I am allowed to execute and what files I will be able to access. So, one way to achieve security is by using something knows as Access Control. So, access control mechanisms in the operating system would allow or manage the various resources. For instance, if I login to a system using a guest account then the operating system will detect this and determine what are the resources and what are the files that I can access.

For instance, in normal guest accounts, I may not be able to access let us say the USB drive while on the other hand I may be have given read only access to certain files

present in the hard disk. While other files which are system related or based on the operating system will not even be given read, write. So, I will not be able to view those files at all. Another technique, which is used by the operating system in order to achieve security, is the user passwords and cryptography. So, passwords ensure that unauthorized people are not allowed to use the system. So, modern laptops also use techniques like biometrics or figure prints scans to filter people who are given access into the system.

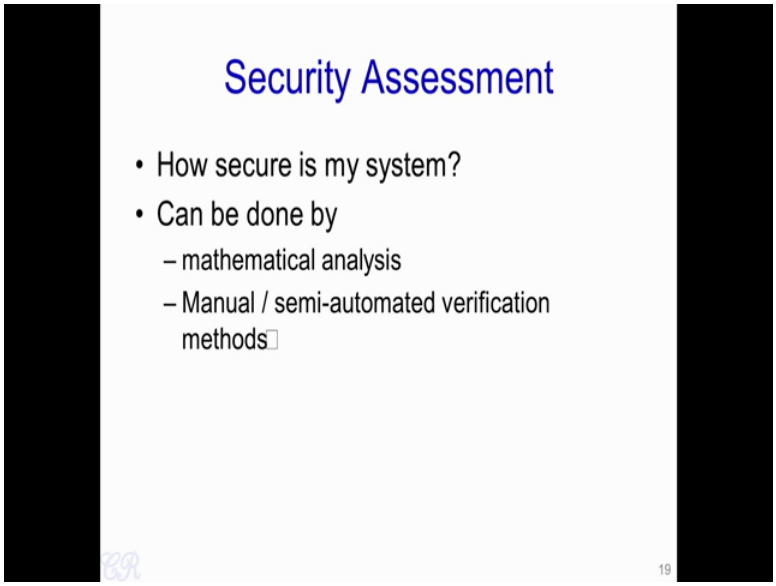
(Refer Slide Time: 32:00)

So this particular slide shows how access control is implemented in typically. So, essentially we have a matrix here (mentioned in the slide above) and each row corresponds to a user. So, Ann, Bob and Carl are three users of the system. While the columns show the resources i.e file 1 which is stored in the hard disk, file 2, file 3 and program 1. So, in the system the operating system depending on the user who has logged in based on their password is given different permissions for each file.

For example, the user Ann has the read and write permission for file 1 as well as file 2 and can execute program 1. However, Ann has no permission at all for file 3. In a similar way, Bob another user of the system can only read file 1, he cannot write to file 1, but he can read as well as write to file 3, and based on permission said in the operating system, bob does not have the access to program 1 that is he cannot execute the program 1.

Similarly, the third user Carl cannot have access either read or write access to file 1 and file 3, and can have only access to file 2; however, Carl can execute program 1 as well as can read program 1.

(Refer Slide Time: 33:46)



So let us say now that we have our system filled with the processor, the various resources and the operating system which manages these various resources and also manages the applications that execute on the system. So, how do we access the security of the system? So, there are two ways in which this is generally done. So this is done by mathematical analysis or by manual or semi automated verification techniques.

So, many systems especially those which are used in critical applications such as the military or other defense related applications, go through rigorous security assessment to ensure that these systems including the operating system as well as the hardware and applications have sufficient amount of security, so that it could be used for such critical applications. So, this is more of a testing and validation related aspect in the system.Thank you.