**Operating Systems**
**Prof. Sorav Bansal**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Delhi**

**Lecture - 01**
**Introduction to UNIX System Calls**
**Part -1**

Welcome to Operating Systems lecture 1. This is an introductory lecture and we will basically in this course discuss about operating systems and how they work and what are the issues when you design an operating system and implement one. So firstly, why is operating systems an interesting thing to learn ? Well, if you look at the history of computing, most of the landmark events in the history of computing have involved operating system either at the center or at the as one of the components of the event.

So, right from 1960's till 2014 operating systems have gone through series of design changes, implementation improvements, and has always been an active area of research. So, to give you a flavor, in the early days of computing an operating system used to control large machines; for example, the IBM mainframes and allowed people to run multiple processes and allow multiple users to share that large machine and get their work done. Today operating systems are present in your personal computers, they are present in your mobile phones, they are present in your cars and in all kinds of embedded devices .

The lots of different open problems that operating system researchers have to face today; for example we are seeing rapid changes in our hardware. We are going from high performance single core computers to computers which have lots of cores. Already are small machines like laptops have up to 4 cores or 8 cores and large machines, larger machines can have up to 80 or 100 cores, that is not uncommon.

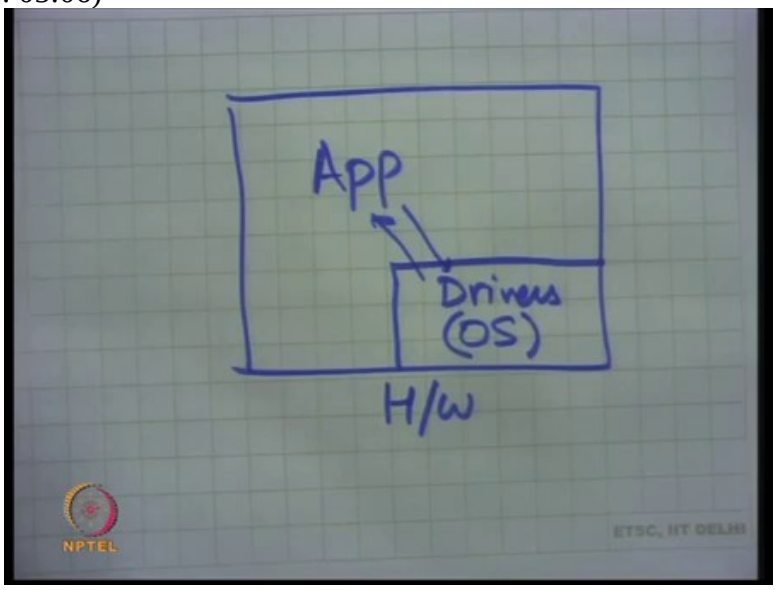
And so, researchers are constantly asking the question, what is the right operating system for this new kind of hardware and do or of current operating systems really fit for these kind of new devices. Similarly we are seeing, advent of new kinds of devices which need operating systems like phones; will the operating system that runs on your desktop, be the ideal operating system for your phone or something better is needed.

Then there are reliability issues; the operating system that runs inside a space mission, let us say the mars mission or something has to be much more reliable than your desktop operating system. And so, reliability is an operating system issue and it is a constant effort of researchers to ensure that operating systems do not have bugs or have ways to prove that operating systems do not have work, so all these are also research issues.

Performance is definitely an operating system issue, I want to write an application and the application is going to run on a certain set of hardware; which perhaps includes multiple multiple processors, each having multiple CPU cores and having memory, disk, perhaps accelerator devices like the GPU's the graphic processing units etc. And how can the operating system allow applications to use all this hardware in the most efficient way. And all these are basically very very interesting software engineering, software design and in particular operating system design problems. So, let us understand what an operating system really is. So, and we will we are going to look and try to understand this by looking at the history of operating systems.

So, the first operating system was perhaps not designed, it was not very ambitious in it is goals; where the goal of an operating system was to allow a program to run . And so, in some sense operating system is the lowest layer of hardware a software that sits on top of hardware. And so, the first operating system would just export a certain set of libraries, that will allow an application to use this hardware. So, the picture would perhaps look something like this.

(Refer Slide Time: 05:06)




You have let us say if I draw the whole system as this box and hardware as the outside area of the box, then the operating systems is basically a bunch of device drivers. And the application is running on top of this operating system and making calls into the operating systems to access the hardware. So, the application runs on top of hardware, but for some operations it can ask the operating system to do it. And so, different applications need to be written for this operating system in this way.

Notice that in this picture, I am only drawing one application running at one time . So, there is only one application that is running at any time on this operating system. And so, this is a uniprocessor operating system or uni process, that's why also called uni processing operating system . And so, only one application can run it at any time and the operating system is just a collection of libraries and which is allowing the application to access the hardware . Let us take a step back and let us also understand how a computer system should really get organized . So, one way to organize an operating a computer system or a software system is to just write all your software as one big program.

So, let us imagine a world where your computer, still the hardware still the same; which basically means there is a processor, there is memory, there is disk etc and they are all interconnected just the way they are today. But the software was indeed written in a one monolithic style, so there is one large program; that is going to do all your things for you. So, for example, this program contains the logic to boot your computer from power of states to something which is usable; it contains all the logic to implement your editor, your shell, your GUI the graphics window manager, your browser, web clients, web servers and all this.

So, this program could be one large program that has all these things built into it. And so, there is this program would probably contain one large case statement, which says if the user does this then go here, if he presses a button here then go there, if he types this command then jump there and so on.

So, you can imagine that all your software on your computer can be organized as one big program that has all these things built into it , it is fact, it is possible; it is not really practical. Because completely different pieces of logic, an MP 3 encoder has nothing to do with the web server, have to be part of the same program and the developers of these two different logics have to now talk to each other, so that they are compatible with each other and can fit inside the same program. Also if there is an update to one part of the program, it basically means updating the entire program.

Moreover there are trust issues; what if I am I want to implement a web server, but I also know that I am sharing this my I will be a part of this large program that also runs the MP 3 encoder. But I do not trust the developer of the MP 3 encoder, because perhaps I do not trust his I do not trust his program; because perhaps here I know that his program may have bugs.

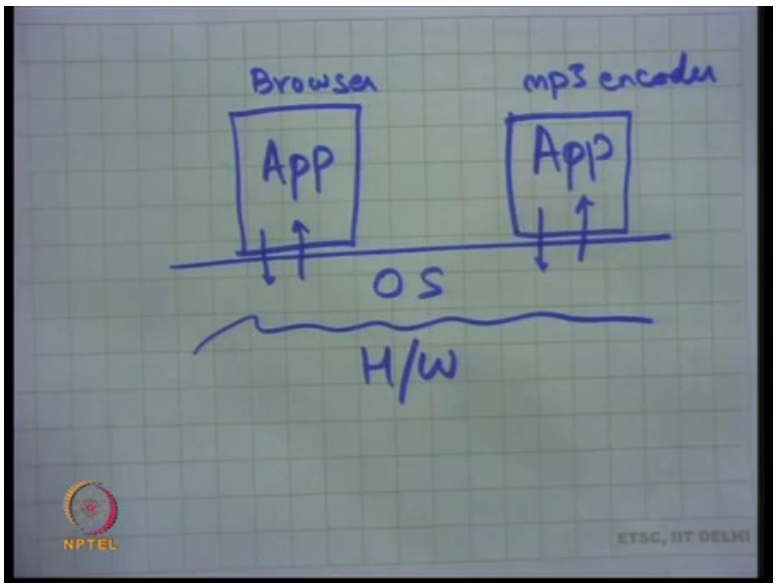
Or I just do not trust the developer basically; that means, that I do not trust him and I do not want to trust him, basically maybe he may want to do malicious things to me. So, all these things are completely not possible in this model where you have one large program. And of course, it has it is also a software engineering nightmare, because if you have one large program that all these things how is it possible to maintain this over a long period of time.

So, in theory there could be an operating system that just implements all the functionality inside it, as one large program and do things. But that is not going to be very practical. So, typically what operating systems do is expose an interface. So, which allow applications to run on top of the operating system. And these applications can be implemented independently.

So, one application could be the MP 3 encoder for example, and the other application could be the web server; and these applications can run can be implemented independently. And at the onetime let us say one application or multiple applications can run together, they all rely on the same interface that the operating system provides. And so, these applications can run on this operating system as long as they obey the interface. In the above particular picture the interfaces happen to be the device drivers.

In yet another kind of interface, you may want that let us say there is hardware, running with my operating system; the operating system exposes certain interfaces and allows multiple applications to coexist at the same time. This is a multi-processing operating system. So, for example, this application could be a browser and this particular application could be let us say an MP 3 encoder or any other such applications.

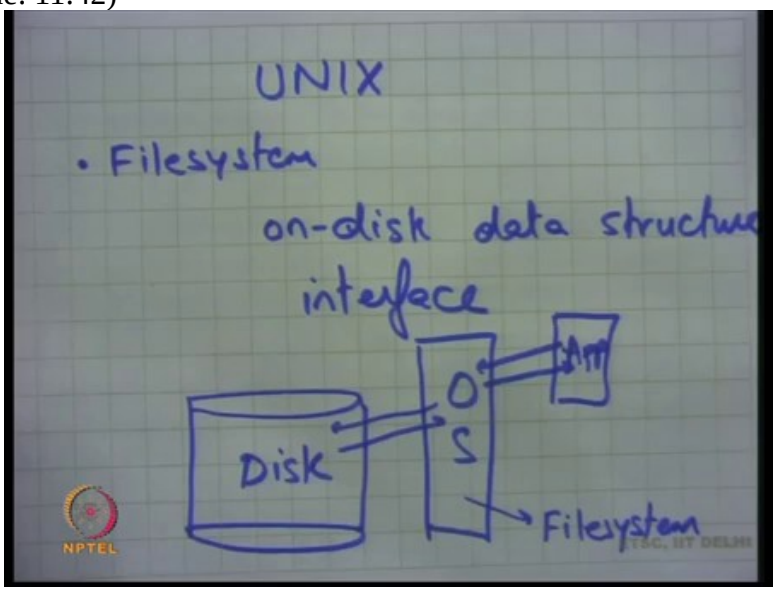
(Refer Slide Time: 10:06)




And now these two, the developers of these two applications do not need to trust each other, they also do not need to coordinate with each other. As long as they meet the specification of the operating system, they can run together at the same time without having to worry about each other . So, the this kind of an architecture is much better for software form a software engineering point of view, it is also much better from a security point of view, modularity and so on and also performance.

So, let us understand what kind of, what are these interfaces , what should these interfaces look like and that itself is when it turns out is a non-trivial problem. So, what should the operating system interface be and let us again trace back to history. So, one of the first operating systems was Multics that was that coming out of bell labs. And one of the successors or Multics was Unix .

(Refer Slide Time: 11:42)




So, there was a system called Unix as developed by Ken Thompson, who was one during award for this for his work on Unix and other people that were involved in this kind this work Dennis Ritchie and others . So, what was Unix? Well, the first version of Unix or the early versions of Unix look something like this, they said I want to be able to run multiple processes on my system and what should the operating system be provide as a minimal sort of interface.

So, they started with first thinking about what. Firstly, what are the hardware components, well the hardware components are there is a processor which we call the CPU, there is memory which is RAM and then there is disk. Disk has the semantics that it is contents are preserved across power reboots, also the disk needs to be shared across multiple processes . So, one process may want to access file a and another process may want to file access file b, they may be running at different times; but the files a and b need to coexist on the disk.

On the other hand at that time, memory could be assumed to be exclusive. So, you could assume that if process a is running, then only process a is using the memory and nobody else is using that memory. Notice that for the disk that is not true . For the disk if , you at the same time it is important that all the contents of all the different processes even if the process is not using them exist.

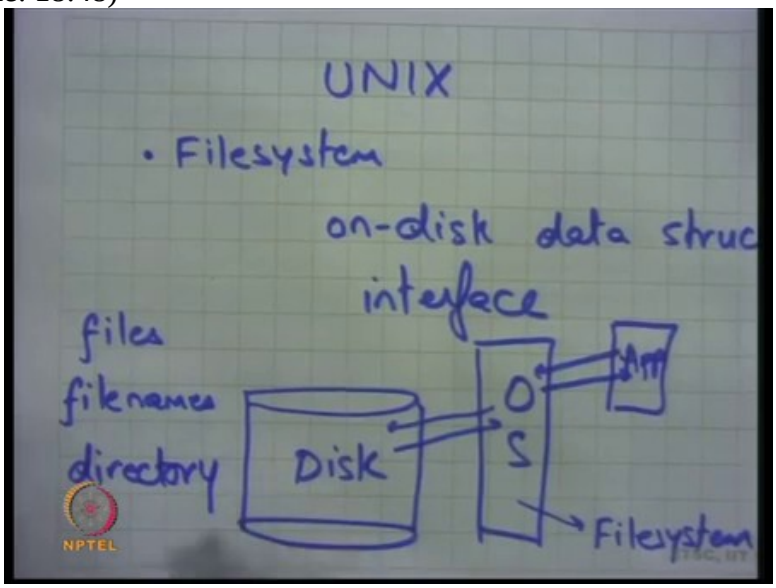
But for memory which is which whose contents are volatile; which means they do not persist across power reboots, at that time they assumed that a process basically has exclusive access to all memory . Of course, today we also do not do that, we basically allow the memory to get shared across multiple processes at the same time.

Well, let us first understand how initially the Unix interfaces look like. So, the other thing that was important was; one important program that was needed was an interactive shell . What is an interactive shell? An interactive shell will give you a command prompt, you will type in your command and depending on the command some program will get to run . When that program finishes you will come back to the interactive shell and this loop and this mechanism will continue forever; that is the minimum that a usable computer system should have.

So, the couple of things that a most very important; firstly, there should have been a file system . So, the early developers of Unix said there needs to be a file system. A file system is an on-disk data structure and some kind of interface to access this on-disk data structure . And this disk has some contents, so there is an operating system that sits between the application and the disk.

And the application is going to make some requests to the operating system and the operating system is going to translate those requests to disk request and then serve the application using those interfaces. And this translation layer was also called the file system. And so, it is clear that any operating system needs to have a file system .

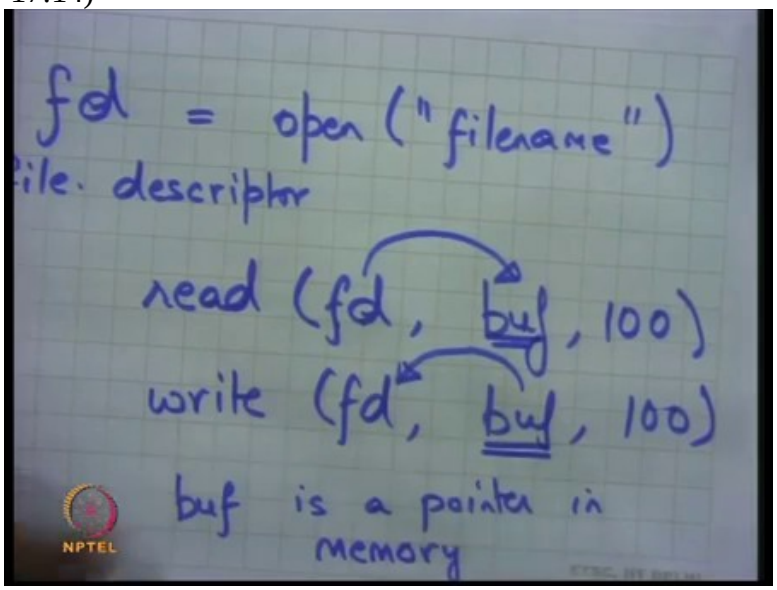
(Refer Slide Time: 18:48)



So, well the file system in early Unix look very quite similar to what we have today; which basically means that files there was a notion of files, which were nothing but streams of characters . And there were a notion of file names . And a process could say, I want to access file name a at offset b and so, the operating system will translate the file name into a disk offset and add offset b to it and give you the contents of that particular file.

Also for better manageability, the earlier file system also had the notion of directories; which basically meant that the files, the file names or the file system or the namespace of the data structure was organized in hierarchical manner. Which basically means that file names were basically had a full path name associated with it; which basically meant where do you go from and so on, which included starting from the immediate parent directory to it is parent to it is parent till you reach the root and so on ok. So, that is one thing that Unix had to have.

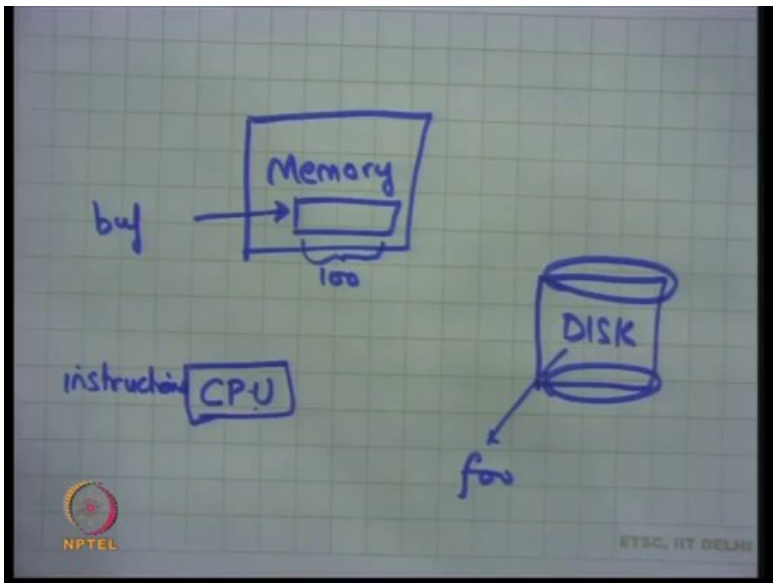
(Refer Slide Time: 17:14)




And the way they did this was basically using an interface which said you open a file; which basically means that I can say I want to open a file foo, then I basically can now use, open up. when I open a file I could let us say open file name and this would give me what is called a handler or a file descriptor which I am calling fd. Then I could read on the file descriptor, I could say I want to read in this file and I want to read from this file, which I have opened previously into a character array buf 100 bytes let us say.

That basically means that I want to read 100 bytes from this file and store those contents into buf. So, or I could say I want to write to this file a 100 characters from buf. So, read basically says, read from file and put it into this character array and write means read from this character array and write to this file. By the way what is buf? Buf is a pointer in memory.

(Refer Slide Time: 18:40)

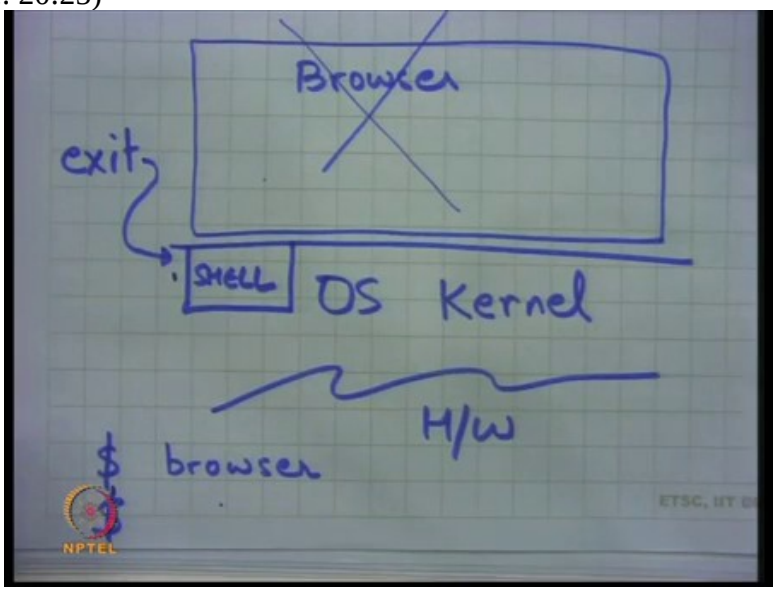



So, if I look at it again, my hardware looks something like this; I have a CPU on which my instructions run, there are memory and disk. So, the CPU is going to run this instruction which will basically make the system, which call this function called read. Buf will be somewhere in the memory and it will have let us say a 100 characters in it and the file foo will be somewhere in the disk and CPU could execute this command called read or write to transfer contents from memory to disk and from disk to memory using these read and write commands.

Notice that the application does not need to worry about which disk it is or how do I access the disk etc; all those things are abstracted away from by the operating system. The operating system knows what disk it is, how to run that disk, how to write to that disk etc. And the interface that the operating system provided or the Unix provided was this read and write calls that the application could make to read or write from the disk. that is one thing.

The other abstraction that they had was that our shell. So, they said ok, because the shell or the interactive shell has to be such an important part of the operating system, the shell was implemented inside the kernel.

(Refer Slide Time: 20:25)



So, let us say there is the operating system OS. And I am going to the operating system is also often called the operating system kernel . So, I am going to use that words kernel and operating system interchangeably and let us say there is the hardware once again.

And one part of the operating system would implement a program called shell . And let us see what the shell does; the shell basically gives you a command prompt, let us say the command prompt is dollar and then you type a command, let us say you type a command browser . So, what the shell is going to do, is it is going to check treat this command as a file name, it is going to search for this file name in the current directory where the shell is running; and if it finds a file name, then it is going to treat that file as an executable program and we are going to run it ok.

So, let us say a browser was a file that existed in the current directory in which the shell is running, then that file will get loaded as an application and control will get transferred to the browser. So, notice that basically the operating system is basically providing interfaces for you to run different programs and allowing different programs to co-exist simultaneously on the disk .

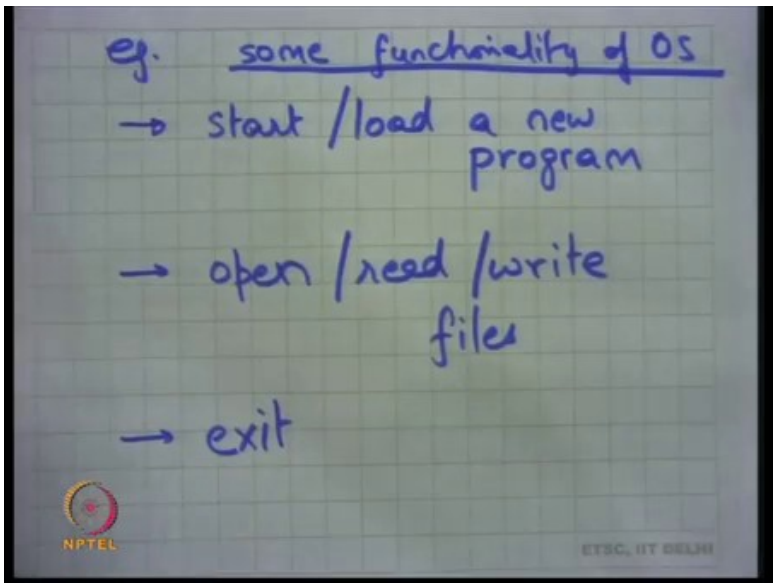
At this point we assuming that, only one program is present in memory at any time . So, and there is a special program called shell inside the operating system or the operating system kernel and this shell is going to take a command from the user; which basically mean it is going to read from the some port let us say the keyboard, so it is going to read from the keyboard, interpret that command; which basically means it is going to typically it will interpret the command as a file name, it searches for the file name in the disk in the file system really . And if it finds a file with that name, then it loads that file name, so that file basically should contain some data, some instructions that need to get executed. So, those instructions get loaded into memory and control is transferred to that particular file program; that program is now going to run all by itself .

So, it is as though nothing else is present in memory, it is just that program that is present. And so, that program is going to run all by itself and when it is running it may make more open calls to open more files; it may make more read calls or write calls to to read or write files and at some point it may want to say I am done, I want to exit .

So, what was how was exit implemented in Unix . So, there needs to be something called exit in early Unix. So, how do you exit? Well, at that time, so in the early version of Unix, exit was basically implemented by just returning back to the shell. so there was another function that the operating system kernel provided which was exit and what that will do is it will remove the occupied process from memory and jump back to the shell to take the next command .

So, you are the print the next dollar sign and here you are and you can now print your next command. There were other things that the operating system had to be careful about. Firstly, if the browser had opened certain files, then when you could return back to the shell; the first thing the shell would do is close all the open files, so that if the new program gets to run, he can open more files and so on . So, this was the simplistic model of the Unix operating system in it is early days.

(Refer Slide Time: 24:44)

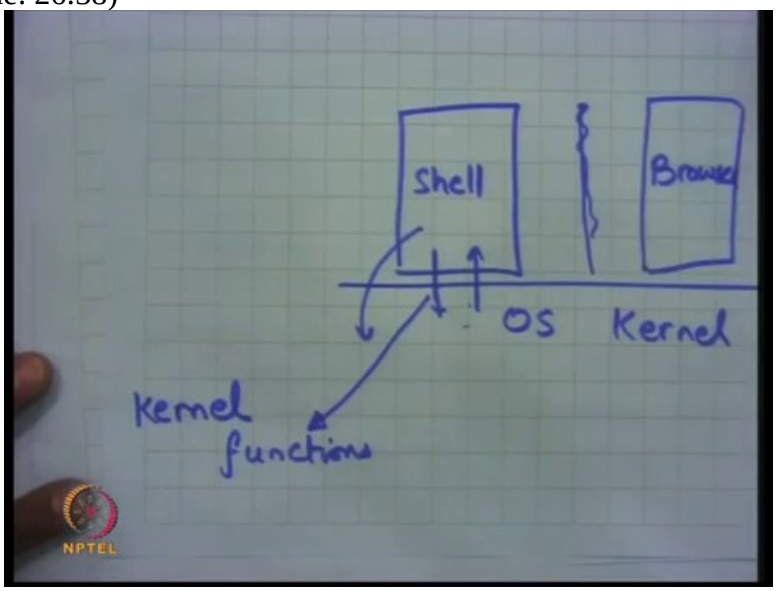



And basically what we have seen is there are few things that the operating system was providing us. Number one, it allowed us to a start a new program, start/load a new program and this it was being done using the special program called shell inside the operating system. It allowed us to open, read and write files and it allows us to exit. So, some functionality of OS that applications use examples. These are example functionality that the application is using and you seen how they are using it .

So, notice that in doing this operating system design, the designer has basically carefully decided that some part of some functionality needs to be part of the operating system kernel. For example, the shell program is part of the operating system kernel and the device drivers are part of the operating system kernel. And some part of the logic does not need to be part of the operating system kernel and it should be present as application logic in executable files; that can be executed by the user at will.

Even very soon people realized that, even the shell has no need to be part of the operating system kernel. One of the important things that go into an operating system design is to make the interface as small as possible and as usable as possible and yet as powerful .

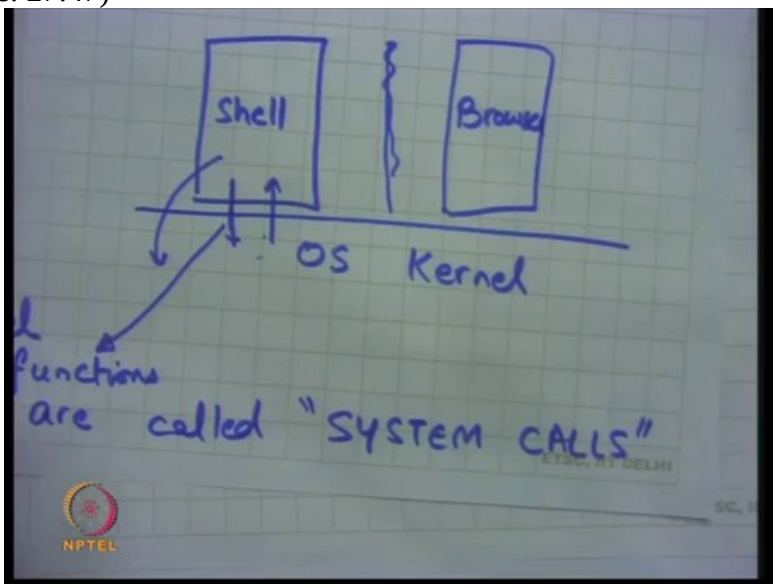
(Refer Slide Time: 26:38)



So, today the operating system kernel does not provide the shell command, instead the shell itself runs as a separate application . And the shell has ways to tell the operating system to start another application. So, for example, the shell could tell the operating system to suspend itself and start a browser; just like before except that this time the shell is not part of the operating system, the shell itself is running as an application.

And in order to do that, it is important that you all have interfaces that allow an application to be able to create another application and jump to it. So, these functions that the operating system provides; these are these functions basically form the interface of the operating system.

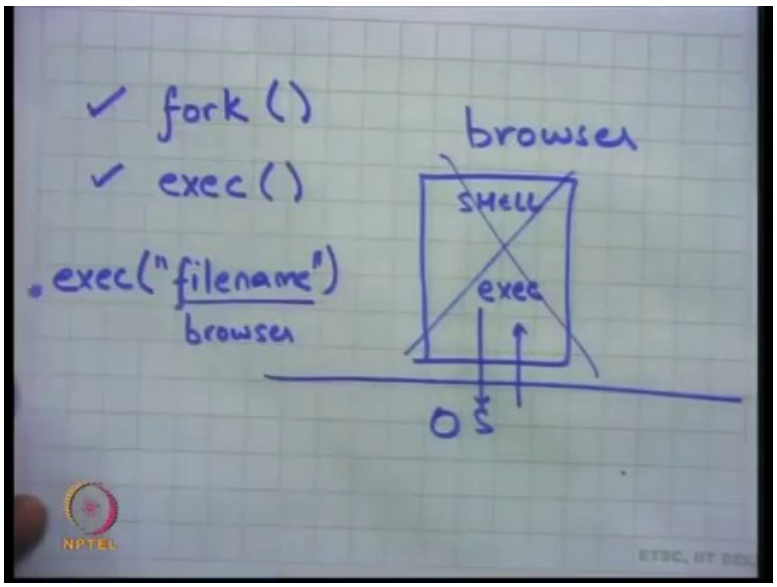
(Refer Slide Time: 27:47)




And let us call these the kernel functions, are called system calls. This is special names to these particular functions that the operating system kernel provides to applications to be able to do things that it wants . And some examples of system calls are calls to allow you to start a new program or to read or write to a file on the disk or to exit .

And there more system calls that were going to look at as we study this course for. So, let us continue with this particular model where we said that the shell itself is written as an application and the operating system provides certain interfaces to allow the shell to start another program. So, what are these interfaces?

(Refer Slide Time: 28:48)




So, Unix provides a system call called fork and another system call called exec and these system calls are used to start a new program. Let us see how fork and exec are used. So, first let us talk about exec. So, let us say I am an operating system and an application is running on the top of the os. Let us say there is a shell running as application program; the shell can make a system call called exec.

So, exec takes an argument which says file name and what happens if a program calls exec system call is that; firstly, the operating system will search for that file name in the file system. So, it searches for the file name in the file system to find in the current directory; to find if there is a file with that file name. If it is there and it is executable, then it replaces the shell with that particular file name.

So, let us say the file name was browser, then the shell will get replaced with browser. So, exec is a way for one program to load another executable or from the disk . And it is done using the file name. So, there is a file name that the program gives that says I want to run this particular program, but the problem is that once you do that you yourself are no longer there.

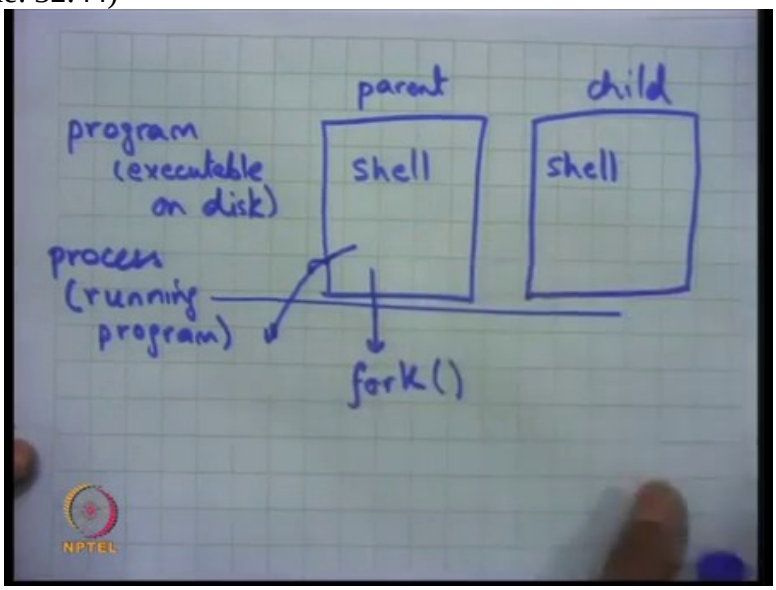
So, at the point when you call the exec system call, you are the one who is occupying memory. As soon as you call exec, the operating system removes you from the memory and instead loads the contents of that executable into memory and transfers control to it. So, essentially that means that if a shell program ever calls exec then the shell program will never get to run again. It will be the new program that will run and when that program calls exit, it is no longer the case that the transfer, the control will get transferred back to the shell unlike in the original Unix .

Because in the original Unix, the shell was part of the operating system kernel and so, it was possible to jump back to it, but here because the shell was an application and the application has called exec. So, it has completely removed itself from the picture . And all it is state has been wiped out and so, there is no way that operating system can jump back to the old program.

Notice that the operating system does not even know that this particular program is the special program called shell. It is just one of the different programs and it does not know what point it was in when the exec system call was called. So, it has no way of reconstructing that program back again. So, exec is one way to load another program, but it also means that I completely get washed out. And there is no way that I can return control to myself later on . So, that is not good enough to for us to be implement to implement the functionality of shell as we know it . Because the shell as we know it basically allows us to type a command, the command gets executed and when the

command exits; the shell can run back again. So, that is something that we want and exec is not going to do the last part which is return back to the shell. So, so the other system called fork can perhaps help. So, let us see what fork is going .

(Refer Slide Time: 32:44)



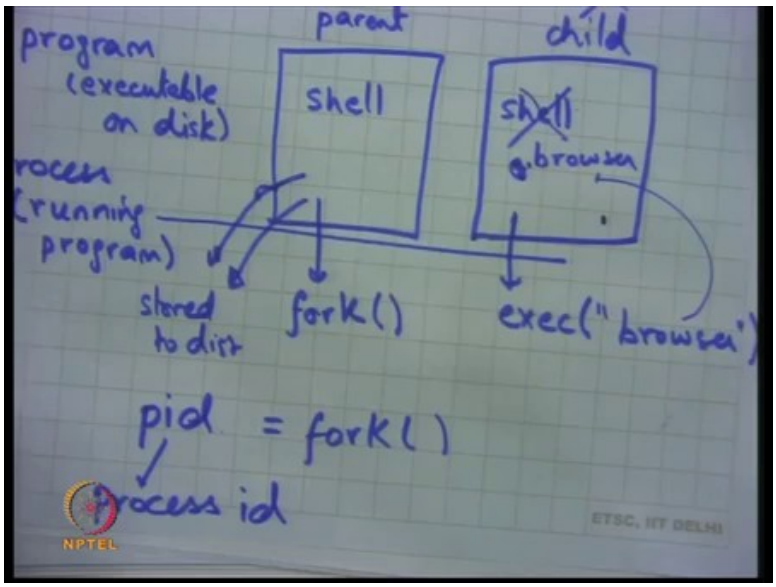

So, what is fork? Well, let us say there is a program called shell that is running. It calls a system called fork; what happens is when it calls the system call called fork, two different apps get created simultaneously identical to each other. It is like a parent.

So, one child one process creates an identical replica of itself using fork. So, they are two shells that get created and at any time only one shell can run. So, the fork system call is going to do is it is going to create two copies of itself with identical state and identical value of the program counter. So, they are going to both be executing the same next instruction when they call fork.

Of course, in our model we are saying that only one program can run it any anytime. So, what will happen is one of them let us say the; so firstly, one of them is called the parent; the one who called fork and the other one is called the child . The program that called fork is called the parent and the program that just got created new or the process. So, that just what created is called the child. So, a little bit of terminology here.

There is a program which exists as an executable on disk and there is a process which is a running program. The exec system call takes a program and converts it into a process; a running program. The process runs and it changes it is own state. So, process is has it is own state in memory and it as a run it changes it is own state and at some point it calls exit let us say. So, initially the shell was running at the process. The shell made the fork system call and it created two processes; the parent process and the child process . Now, at any time only one process can be active. Let us say and if it is a uni processing system.

(Refer Slide Time: 35:47)




Then, let us say the child process gets stored to disk and the as the parent process gets stored to disk and the child process gets to run. And so, that is the semantics of fork, but our real goal was to be able to implement the shell functionality. So, what can be done is that the parent process creates a child process and the child process now calls exec . So, what happens is the parent process still exists exactly at the point where it had forked the new process. The child process can now load the new program which basically means the shell gets wiped out and instead gets replaced by the browser .

The browser can get to run and at some point, the browser will call exit in which case the whole child process will get wiped out. So, we have seen that there is a system call called fork that allows you to create a new process, which is a replica of the process that called fork and there is a system call called exit, that allows you to stop the current process or completely free the current process and there is a system call called exec which allows you to replace the contents of the current process with the new program. Now using these two system calls; fork and exec, it is now possible to implement a shell like program as an application.

So, for example, a shell will fork first call fork, so it will create two copies of the shell. The child copy of the shell will call exec on the command that you gave, on the command prompt. The command will get to run, when the command finishes it is going to call exit. So, that process gets wiped out. The old shell, the parent shell, that was stored on disk gets loaded into memory and continues from where it left off .

So, that is , that is one way that you can implement the shell functionality. So, as opposed to having the shell as part of the operating system kernel, it is a much more modular way to actually have the shell also as a separate program; much more powerful way instead in fact, to have the shell as another application and have these special system calls called fork and exec that allow you to implement the same functionality as the shell.

In fact, the system calls fork and exec can be used by any application . The shell is just one of the applications and the kernel does not even know that there is a shell . It can be just any other application. So, any application when it calls fork and creates replica processes of the parent process and any process that calls exec is going to load the file into the current process .

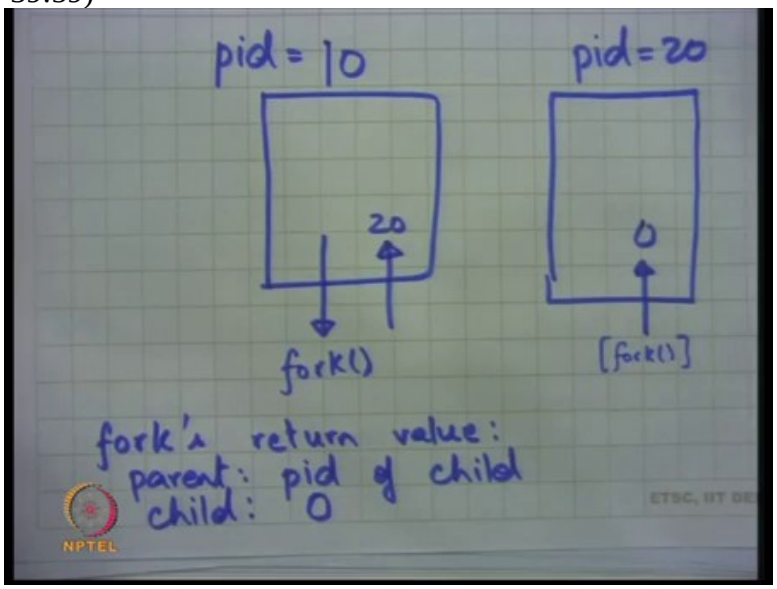
There are there is the few things that I have skimmed over; one is when a system call, when a fork system call is called then there are two processes that return from the fork system calls. So, fork is a

system call that is called by one process, but returns in two processes. So, one process calls fork and two processes returned from folk at exactly the same program counter .

But there was a difference; I said the child will execute the exec system call next, if it wanted to implement the shell functionality; while the parent will just print the next command prompt. So, how do you do this? Because the two programs are completely identical, they are twins; how do you decide whether I am the child and I should call exec or whether I am the parent and I should actually display the command prompt?

Well, so, the fork system call is has a return value. So, the syntax of the fork return call is; fork returns a return value which I call the pid or process id and the return values are different in the parent and the child. So, let me just draw this as a picture.

(Refer Slide Time: 39:59)

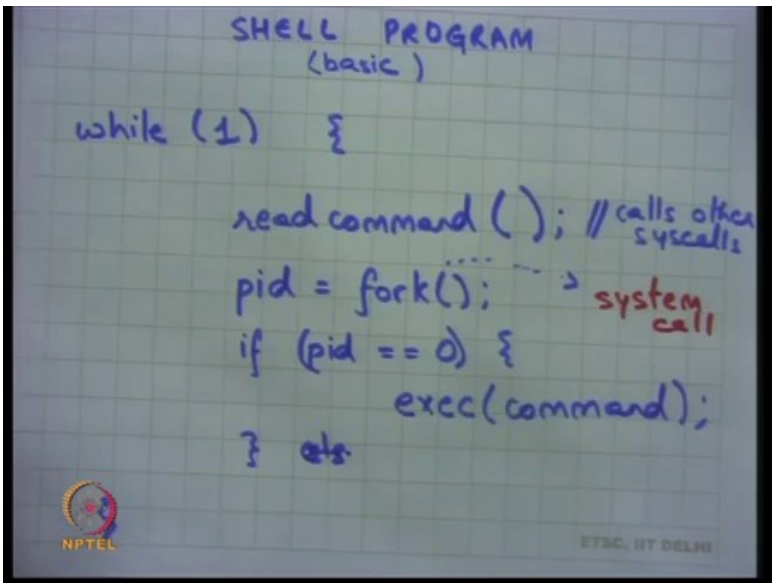



So, let us say thre is a process. So firstly, every process has an id. So, let us say this is process 10 which basically means each process has an id. So, this process has an id of 10. So, pid is equal to 10. This process makes a system call called fork and the operating system returns from this system call, but it returns in two places.

So, it creates another process, let us say it creates another child process and this child has a pid of 20. Pids are assigned by the operating system. So, it can choose any way of assigning opponents thing and this one also returns behaves as if it has just returned from fork . This, particular process did not ever call fork, but it behaved as though it has just returned from fork. Both of them are going to return at exactly the same program counter except that the return value in the parent will be different from the return value in the child,.

So, in the parent the return value is the pid of the child. So, the return value in this case will be 20. In the parent process, the return value is pid of child and in the child process it is some number which cannot be a pid, let us say 0. So, here are the return value is 0. So, all that you need to do, all other all that the programmer needs to do is to check the return value. If the return value is 0, then I will call exec; if the return value is non-zero, then I will print the next command prompt for example .

(Refer Slide Time: 42:09)




So, let me just write this more formally. I am going to write the shell program, very basic it looks something like that shown above. It is an infinite loop(while(1)) just continue still the user types exec or just some something else to basically stop it, but otherwise it just keeps running. And it has some function to read command, how it reads the command let us just abstract it out. Let us say there is some function which says read command . It is going to make some system calls internally to basically read the command. This read command is not a system call; it is I am just using this function as a placeholder to say it will call other sys calls that we will discuss later .

And, it will say pid is equal to fork. This is a system call. Let me write it in another color system call . And then I am going to check, if pid is equal to 0 then exec command, whatever you read from here. Else, let us say else you just read the next command. So, what is happening; here is the program that will at a very basic level implement the shell functionality. It reads the command, calls fork. In the child process it calls exec and in the parent process it just reads the next command . So, this will exactly do what we wanted it to do basically; which means it will execute the new program and allow the previous program to do this to repeat the same thing again.

But the previous program will probably get to run only after when the new program has finished for example . So, this is sort of uni processing environment where only one process gets to run and let us assume that the child process gets to run first the child process is going to run and when it calls exit then it is going to go away and the parent process is going to get to run now.

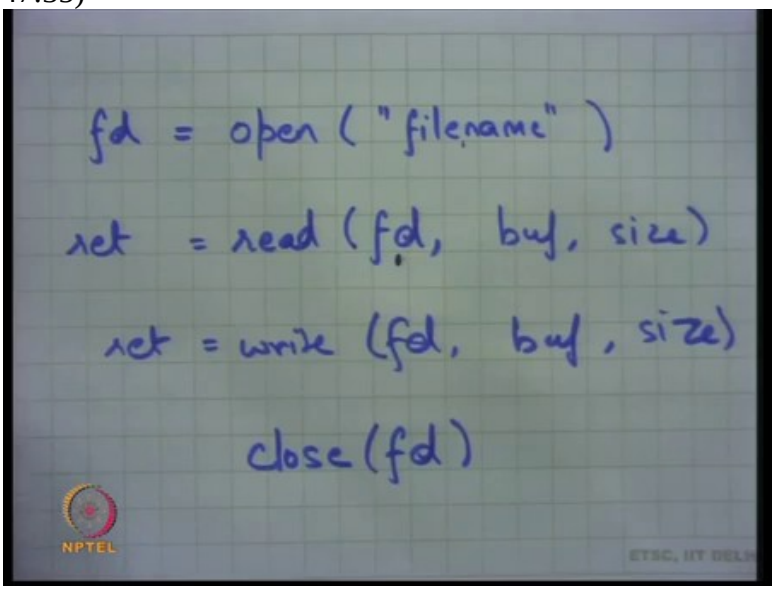
And the parent process is going to go back in the loop and read the next command . So, that is what is happening in this loop. So, there are many details or many things that I have omitted here, but it is important that before we go there you understand the functionality of fork and exec and the use of fork and exec.

Now, let us understand how do you read the command . So, typically we think of read command as something which is being read from the keyboard . So, the user is typing some command and he presses enter that is a command. So, how do you how do you allow an application to read the key is being pressed on the keyboard . Similarly, how do you allow an application to be able to write characters on the console. So, both these things are mediated by the operating system and in general any resource that is a shared resource which needs to be shared across multiple applications is usually mediated by the operating system kernel .

Because if anything is shared then the operating system needs to make sure that there are no concurrent accesses by multiple applications; there different applications are not stepping on each other stores for example. So, operating system need to mediate in the middle. Some examples are, we just saw that the file system is a way to for the operating system to allow mediation of the disk blocks .

So, the file system is basically in on disk data structure; that is shared by multiple processes. And the operating system basically exposes functionality like open, read, write to allow applications to access this these disk blocks which are shared. Similarly, the keyboard and the console and that the other hardware devices are shared resources and different processes may need to access these shared resources and once again all these hardware devices, the access to these hardware devices is mediated by the operating system. One contribution of the Unix system was the general interface to be able to use across a variety of devices and other shared resources.

(Refer Slide Time: 47:35)




And this interface is open, read, write and close . So, I can say open a source which and I named the resource using a name . And it will let us say return a file descriptor. So, we have seen this before, but we are seeing this in the context of files only, but the same thing can be applied to other things like devices, like the keyboard and the console and how we are going see very soon.

So, when an application says open, that is the time when the operating system is going to check whether let us say somebody else is using that resource. If so, perhaps operating system; you want to say no I will not allow you to open it in which case the open can return an error .

So, typically these system calls return positive values and if they return a negative value like minus one; it indicates that the system called failed. So, if an application calls an open system call and it gets the minus one return value, it basically means that the device was not the file or the device was not opened. Once you open, if let us say you opened a file name and a file descriptor was returned which was a non-negative value, then that file descriptor can be used to as an argument to future calls which are read.

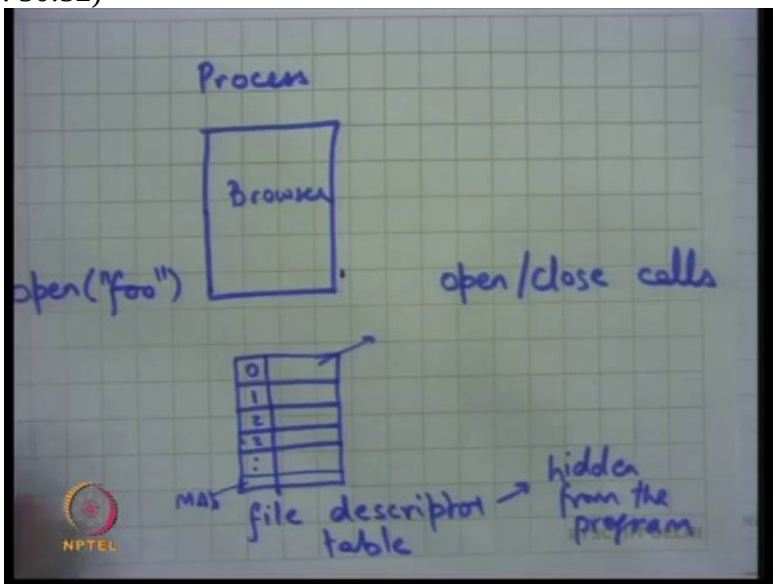
we have seen this before read buf size, read and write to that resource. Once again that the operating system can decide whether it allows read on this particular file descriptor or not. So, for example, if I try to read from the console, I should get an error. So, it has a return value which can be which can

be negative which basically means over that you are read was illegal. So, I should not be allowed to read from something which is an output device or output file.

Similarly, I I should not be able to write to let us the keyboard or to a file which is a read only file and so on. So, the nice thing is because of this interface, the operating system gets to also in check or do perform access control for the applications to the shared resources. And then finally, you could do closed file descriptor to close that particular file; which basically means that I am done with this particular file and other people can let us say use it etc.

So, it allows the operating system to know who has which files open . So, let me take this a little further. So, how is the file descriptor returned? It turns out there is also a way, there is also a convention in which these file descriptor numbers are returned. So, whenever the file, a process calls open a certain file descriptor number will get returned and let us understand what this convention is.

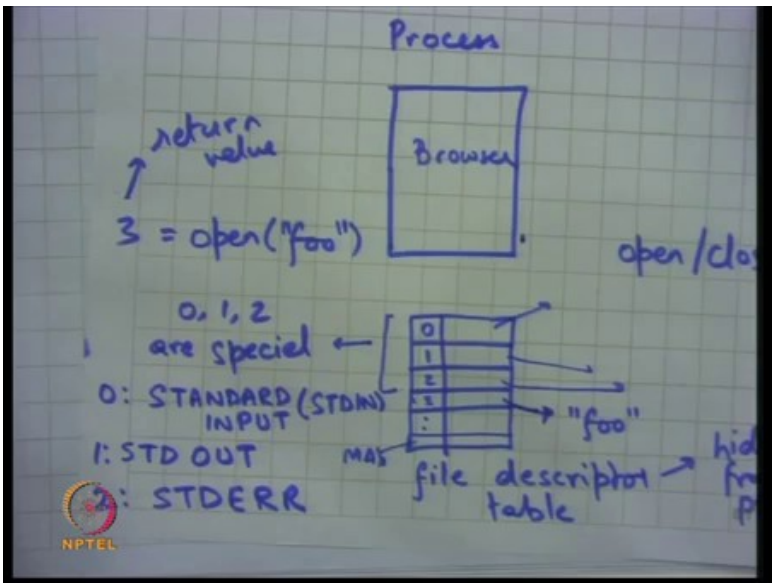
(Refer Slide Time: 50:52)




So, we said there is a notion of a process . A process is a running program. So, let us say I am drawing a process as a strict angle and there is a program that is running inside it, let us say the program is browser . Each process also has certain state called the file descriptor table. And, this file descriptor table is hidden from the program. I mean the program cannot access it directly, but the program can manipulate it using open and close calls .

So, when a program calls open; the operating system searches for the first available file descriptor and so, let us say I say open foo. I search starting from the beginning. So, the file descriptor table has key value pairs, the keys are numbers starting from 0 till some maximum value. And the values are structures which contain whether it is opened or not, and if it is open what are it is weight specifications.

(Refer Slide Time: 52:33)

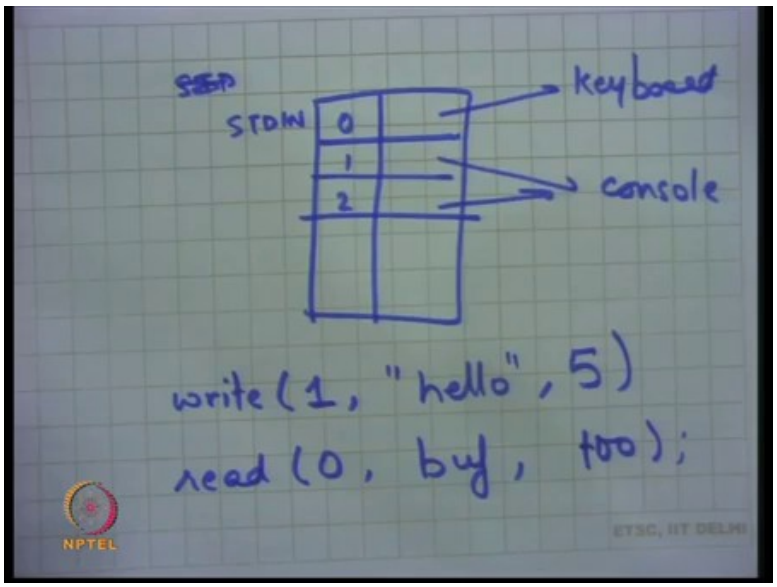



For example, 0 could be pointing somewhere, 1 could be pointing somewhere else, 2 could be pointing yet somewhere else and 3 may be empty. So, when I say open foo; what will happen is the operating system will go through the file descriptor table, look for the first unused fd which is 3 and it is going to put foo here and return 3 .

That will be return value to the application. The application can then use read, write, close on this file descriptor to read or write to this file as we saw them. The first three file descriptors 0, 1 and 2 are special. They are called standard input or STDIN, standard output that is 0. This 1 is standard output STD OUT and 2 is standard error. So, process has three special file descriptors 0, 1 and 2 which are, which referred to it is standard input or their special names for these three file descriptors and the names are standard input, standard output and standard error. The idea is that a program typically will read from its standard input and write to its standard output and if there is an error, it will write to the standard error output .

So, if I am a shell program then I will always read from my standard input and write to my standard output. I do not care whether the standard input is the keyboard or something else. I do not even care whether the standard output is the console or something else. The program is written, the semantics of the program that it reads from the standard input and writes to the standard output. And, somebody else let us say the operating system is responsible for deciding what is the standard input for this program and what is the standard output for this program . So, let us assume that the standard input.

(Refer Slide Time: 55:14)




So, let us assume that the standard input. So, zero or standard input is points to the keyboard and standard output points to the console and standard error also points to the console . So, now, if the program calls write(1,"hello",5) i.e, write on file descriptor 1; basically, says I want to print this on buffer which has five characters in it, then that program is effectively writing the string hello on the console . If the program says read from standard input, some characters into a character array buf which he which he may have declared of some size let us say 100, then it will read a maximum of 100 characters, but if the user pays 100 at enter before the 100 characters that is also fine.

So, it is going to read some a set of characters from the keyboard from to this buffer . So, that is the way you read or write to or from the console or the keyboard . And, notice that very very elegantly and very smartly the operating system designer has used the open read write closed system calls to also be able to read or write to the devices and not just to the files. So, we are going to continue this discussion in next lecture and yeah that is it for today.

Thanks.