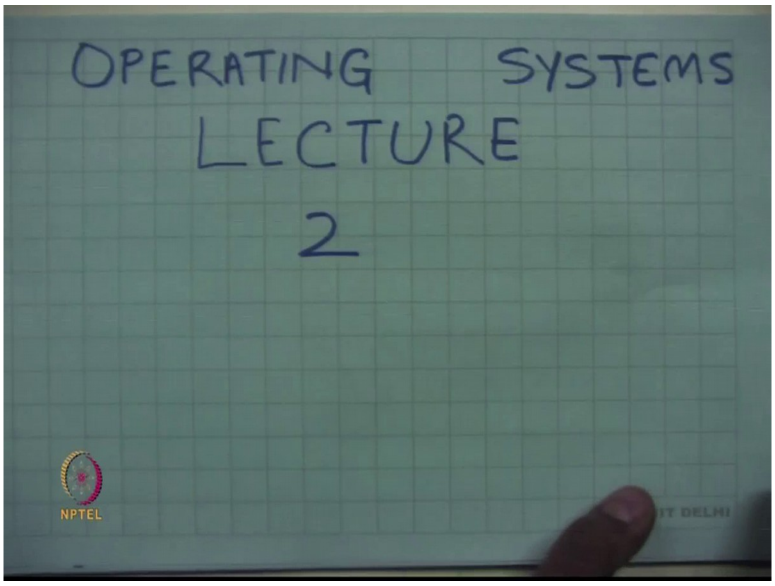
Operating Systems

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

Lecture - 02
Introduction to UNIX System Calls Part – 2

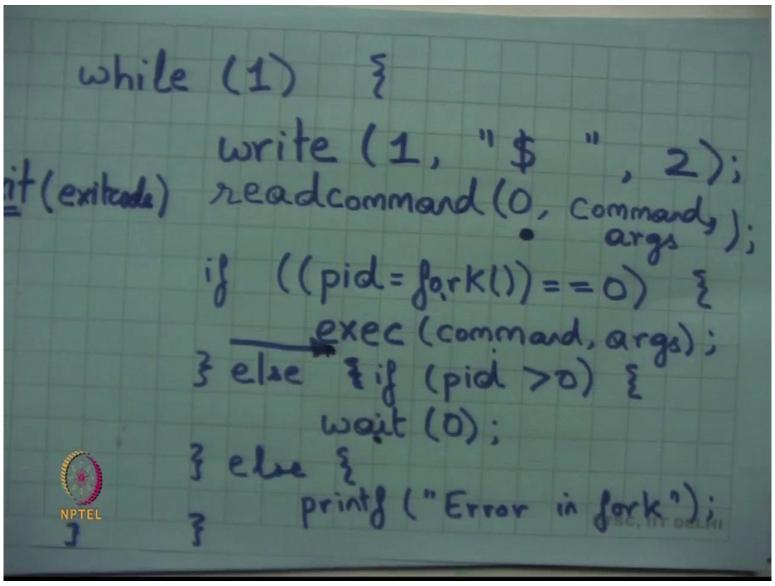
(Refer Slide Time: 00:25)



Welcome to Operating Systems lecture 2. Yesterday, we discussed that software for computer system is usually very complex, it involves lots of different things including event handling for devices, running different programs, loading different programs, writing new programs and so on. And its important from a manageability and point of view to structure or your system in such a way that it is; it becomes more manageable and that is the; that is in some sense the area of operating systems.

And we discussed then an operating system is in some sense the lowest layer of software that sits on top of hardware and it exposes certain API or interfaces to the applications so that they can use those interfaces to do all the things that they would want to do. And we said that in general designing such an API is not revealed; its little complex and we started with looking at one particular operating system Unix which was one of the first very successful multi processing operating systems and on which many current operating systems are model and we started looking at what its interfaces .

(Refer Slide Time: 01:47)



```
while (1)   {
        write (1, "$  ", 2);
it(exitcode)  readcommand (0, command,
                                args );
        if  ((pid = fork()) == 0)  {
            exec (command, args);
        } else  {if  (pid >0)  {
            wait (0);
        } else  {
            printf ("Error  in  fork");
}       }
```

In particular, we started looking at one powerful program called shell which lot of us use in daily life and see how a shell is implemented on top of Unix subtractions . And so we said look the shell is actually implemented as this loop which is denoted by the while statement. In this loop, the first thing it does is write to its file descriptor number 1, a string of length 2 that is the dollar form that you see on your screen.

The file descriptor 1 is a special file descriptor which points to the standard output, the standard output could have been opened by the shell itself for example, or it could have been inherited from its parent or whatever other ways this the file descriptor could have been initialized. Moreover this file descriptor could point to a file, it could point to a device, it could point to the network so that allows you to write really environment agnostic code when you are coding something like that. So, the same shell can execute both on a single code and can execute on your screen.

Then there is this is imaginary functions that I have written here it is called read command you can; so what this read command function does is, it invokes the reads system call . Once again as a reminder a system call is the function that a kernel provides to an application . So, at inside the read command function it must be calling the read system call and it must be it and its calling the read system call on file descriptor 0 .

The file descriptor 0 is also a special file descriptor which refers to the standard input of the program and once again, it can be initialized in any way exactly as it was initialized for the standard output. It reads commands from the standard input, parses them into the strings called command and args. You can imagine that this command is in an array of characters and an arguments is also an array of strings where each string is an array of characters.

So, once it has parsed the command which it has read from the standard input of the shell, it needs to actually execute that command . So, let us assume that the name of the command can be parsed as a name of file on the file system and so what you are going to do is you are going to execute that file, you will going to pick up that file from the file system and you are going to execute it.

But once again you cannot just execute a file in the context of your own process. The shell is running in a certain container which we call the process which is again an abstraction that Unix provides and so the way it works is that Unix allows you a system called fork, which clones the current process. So, which means it makes a copy of itself and creates another process which we call the child process. The new copy of the process is identical to the parent process except that its process id is different and the return value of the fork system call is different .

So, we assign the return value of the fork system called to this variable called pid which is one local variable and then we check its return value if that return value was 0; it means that we are currently executing this code in the context of the child process; in which case we execute another system called exec which takes the command and the arguments and tells the OS to execute this command in the context of the child process .

So, the parent process remains completely unaffected by this operation of exec which executes only in the child process. The semantics of the exec system call are that it will completely forget about the current process in the terms of the memory contents; load the executable into the context of the current process and started running .

So, in other words; once you call the system call exec, control actually never reaches the point indicated in the above picture(just after exec system call). Because once you have called the exec system call you have loaded the new executable, you have transferred control to the first instruction of that executable and that executable has completely forgotten the context in which it was loaded . So, the process is completely sort of reinitialized in some sense.

So, you could write anything there, but it will not get executed. That is the semantics of exec never returns, but an executing process or any process can exit . So, there is another system called exit which , this is again a system call or a function provided by the kernel; that a program that a process can call and the semantics of the exec system call are that the process will terminate and all the resources belonging to that process will get read .

So, all memory etcetera will get read, whatever belongs to that one. So, that is the semantics of the exit system call. Often, when a parent spawns or forks a child process; it was also interested and what happened inside the child process whether the execution of the child process exceeded, whether it failed or what was it return value for example .

So, Unix allows you to specify an argument to the exit system call which let us call the exit code or you can which is just an integer which can be thought of as a return value of the process as it exits ok. And the parent; so that is that is how the child executes, it calls the exec system call and the exec system the program that is loaded inside the exit call system call may or may not call exit at some later point of time.

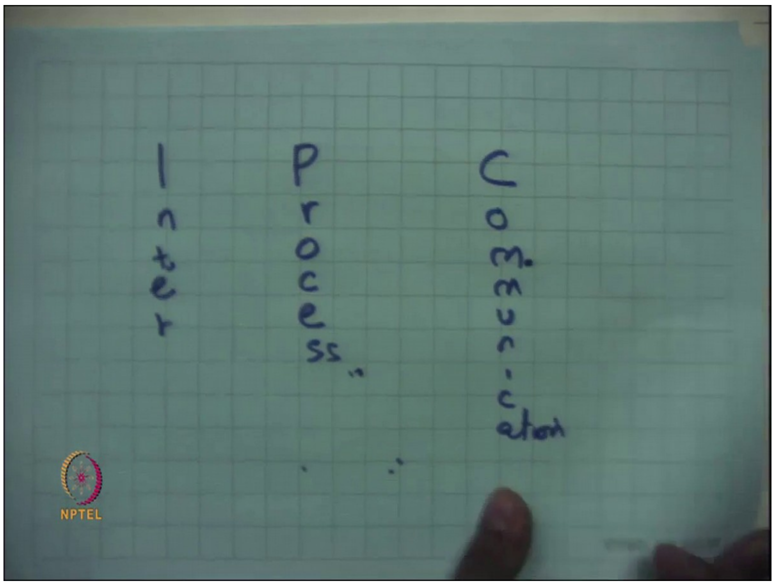
The parent can monitor what is happening to the child by calling the wait system call . So, the semantics of the wait system call are; in this case I am calling it with the argument 0, it basically means wait till any of my children exists . So, as soon as that child; so in this case a parent will only have one child process at most living in the system . So, the shell process is one program; one process and you will only fork and then you will wait.

So, at most you will have one outstanding child in the system and so when you calls wait 0; you basically waiting for that particular child that you just want. So, the semantics of wait is that will block till the child is running and it will return as soon as the child exits. Further, the return value of wait is the exit code that was passed as an argument to the exit system call in the child .

So, if the child called exit with return value 10; then the return value of the wait system call in the parent will return 10. So, that is the limited way of inter process communication between the child and the parent , it is a question. So, what happens if the child does not call the exit system call, but crashes for example, it touches illegal memory and call the segmentation fork . Even in that case the wait system call returns with the appropriate error code, indicating what happened to the process.

So, either either you will get the exact code in inside the return value or you will get us get a special code which indicates that the process crashed and why did it crash. The parent can get that kind of information for a child; it is a limited form of what are called inter process communication.

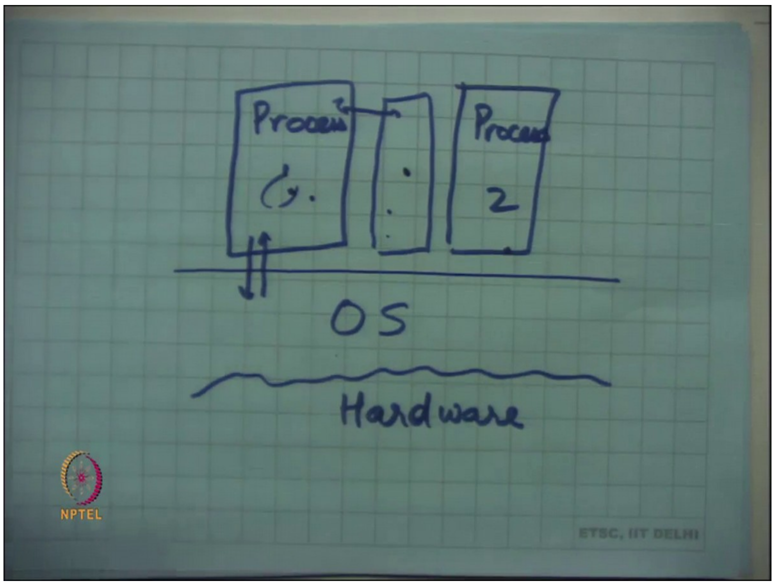
(Refer Slide Time: 10:07)


Inter
Process
Communication

I P C; Inter Process Communication.

Once again, if I have to draw the diagram of the OS, there is the OS, there is let us say the hardware and there are different processes process 1, process 2 and so on . These processes are making system calls to the OS and getting answers to the system call.

(Refer Slide Time: 10:21)



One of the system calls is the fork system call which actually creates another processes such that it is identical to the original process. These processes are more or less isolated from each other. So, if I am if I instantiate a variable in some process and I write something to it, it is completely in my private space as opposed to a variable that is instantiated in other process.

However, often you need to communicate between these processes for various reasons and one way to communicate is what we saw is the exit code between the exiting process and the waiting process. This form of inter process communication is very limited because it is just only be used between a parent and a child, we cannot use it for any two arbitrary processes; also it can only be used if the child exits. So, it is a limited form of inter process communication and it only is useful when one of the processes are actually exited .

We are going to see more types of inter process communication very soon. Anyways, so exit code is going to tell you what happened to the child process. And most shells would typically have an option which say whether you want to display the exit code with which you can find actually whether your command succeeded or failed.

One of the standard conventions on Unix for example, is that if a process execution succeeded; then it will exit with exit code 0. If it failed then it will exit with exit code 1 or it can or some nonzero exit code indicating the failure condition. And the third if condition in the above picture is what; so if the return value of fork was 0 which means it is a child.

The second one was if its greater than 0 which means it was parent and the other the remaining one is when its less than 0; which indicates error in the fork system calls; the fork actually never happened . If the return value of fork is less than 0, it basically means that the fork never happened for whatever reason ; it can be many reasons for example, the process ran out of memory. So, this kind of interface though seems rather easy now was actually not so trivial to come up with when it was first in such .

So, there were many iterations before of what this interface should be before Unix was actually became popular. And as you can see this interface of; so this interface really involves something like a fork system call and exec system call, the ability to name file descriptors and to be able to call open, read , close on these file descriptors. The exit system call which returns an exit code and the wait system call which allows you to wait on an exit code   and all these things what they allow you is composability.

So, it allows you to compose one program inside another program, able to spawn another program and treated just like a function call for example. So, you just one another program let it run; let it run asynchronously, collect its return value; it sort of becomes an asynchronous function call. You can also pass arguments to that one process.

So, specific arguments with that spawn process by saying what the file descriptors are going to be initialize the standard input, standard output, standard error; error based on what you want it to be and that sort of gives you a very nice composability, composable setup and allows you to treat programs as tools which can be composed and fitted in any place in an environment that you like to. So, in that sense it was; it makes things very clean and the and useful and in fact most operating systems today some flavor of this kind of an interface.

We also saw how if I wanted to redirect certain file descriptors to other resources than what shell is currently using and I could do that by inserting appropriate code before the exec system call inside the child container . So, for example, when I fork something, the semantics of fork are that the entire file descriptor table of the parent gets copied into the child .

But, if I wanted to say; I want to redirect the output of my child process to some file let us say, then what I will do is I will close the existing standard output and then I will open the file and the once again the semantics are then open, will return the first available file descriptor in the file descriptor table which means standard output will get reinitialized to file.

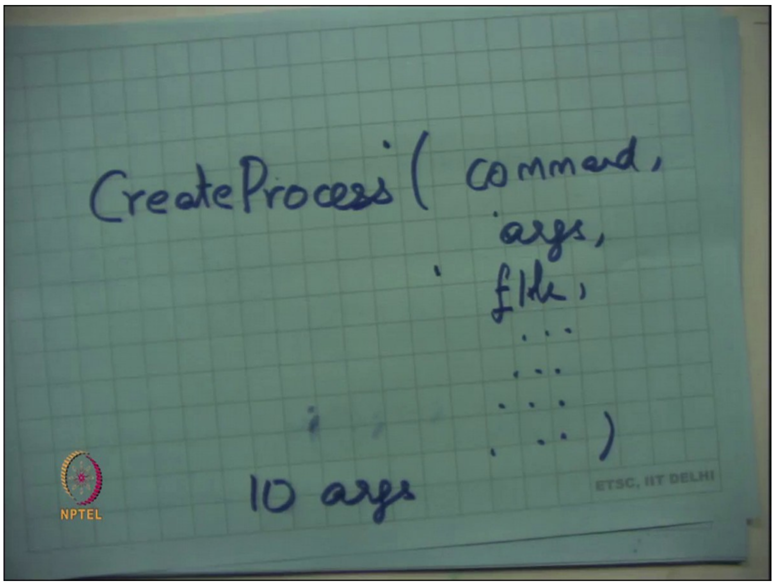
And now exec of ls can take place notice that in this I dint have to change the code of ls at all; ls was its composable in that sense because I just changed the environment and then invoke ls with that new environment and now ls behaves as exactly as I wanted it to be; This is one interesting example of composability.

Notice that this is possible this; this way of redirection is made very easy because I have separated this process creation into two system calls. One is the fork system call and the second is the exec system call and now I can do some things in the middle between the fork and the exec system. It is not very clear why this is a very very interesting choice really. Because you may argue well I am it is a little peaceful to do things in this way.

Because I first create a copy of myself using the fork system call and then overwrite the copy with something else . It seems wasteful that I first copy the entire contents and then completely overwrite it with something else. So, the first copy was really not needed and why am I doing it; I am doing it because Unix allows that is the only way Unix allows me to create a new process. So, there is the seems; it is like it is a performance problem.

However as I have discussed so far it gives you a lot of flexibility. Because you can do things between fork and exec. In fact, Microsoft windows does not do things in this way ; so, it does not do fork and exec.

(Refer Slide Time: 17:27)


CreateProcess(command,
args,
file,
. . .
. . .
. . .
. . .)
10 args

It has a system call called create process. So, I guess the Microsoft windows engineers; OS engineers felt that the fork system call is too costly and I we do not really need to do this and so they have a system call which is create process which takes; let us say a command and arguments and the semantics of the system call are that it will start a new process and start this command in that new process. So, there is no fork ; so there is no copy being done and so you avoiding that wasteful operation, so called wasteful operation on Unix .

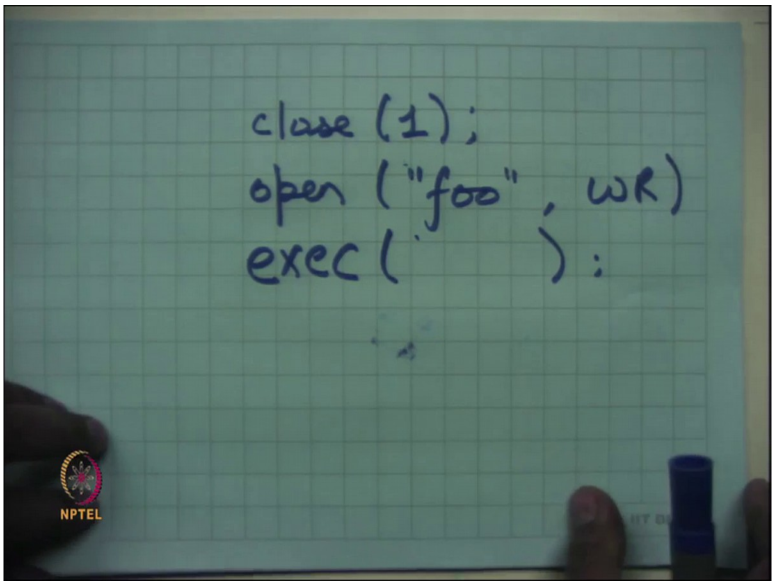
However, as you can imagine; the new command may need different types of environment. For example, if I wanted to implement shell redirection on something like this; there is no there is no clear way of how to do it . So, if I wanted to say ls greater than file; who not quite possible unless I change the code of ls. So, what they did was they added more arguments here, like file descriptor file descriptors and so on environment variables and so there are roughly; 10 arguments in this system call.

So, window system call says create a process with this command and these arguments and this environment . So, that makes your system call rather bulky, but arguably it perhaps more efficient than Unix ok. So, there is a; there is an interesting trade off I mean here is an example of a trade off between clean interfaces, small interfaces and performance interfaces.

It turns out that Unix in its fork and exec is not all that costly ; as we are going to discuss later in the course, it is possible to implement fork in a very fast way . Using what are called copy on write optimizations where when you create a new process, you do not necessarily copy the entire process up front, but you just create pointers to the original process with read with and set the entire process read only in read only mode and only copy when something is written in the new process.

In the common case when the new process actually just exits to something else, your fork was pretty much free . So, interesting example of 2 D algorithm operating systems doing the same thing in different ways ok; last time we saw how shell implements redirection .

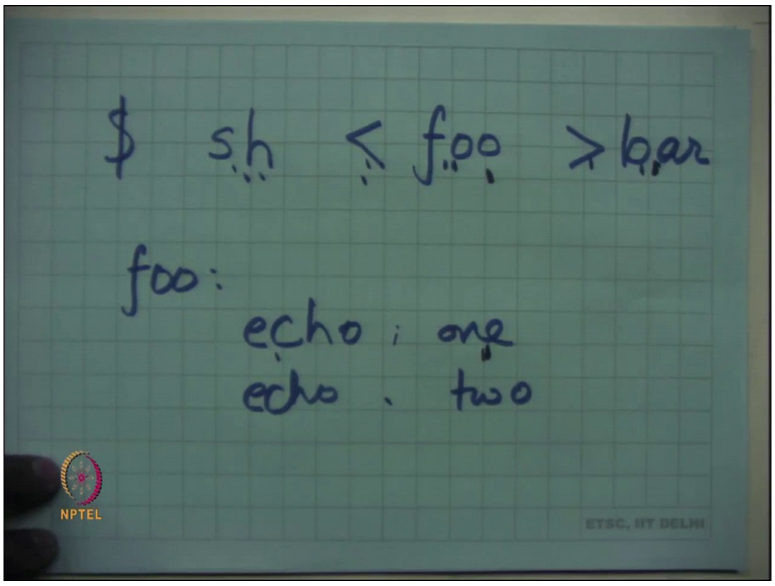
(Refer Slide Time: 20:12)



```
close (1);
open ("foo", WR)
exec (        );
```

So, the way it implements redirection is that before it calls exec; it closes the standard output, close 1 and then let us say opens foo and let us say write mode and then calls the exec whatever command there is and so that gives you redirection of commands on the shell .

(Refer Slide Time: 20:43)



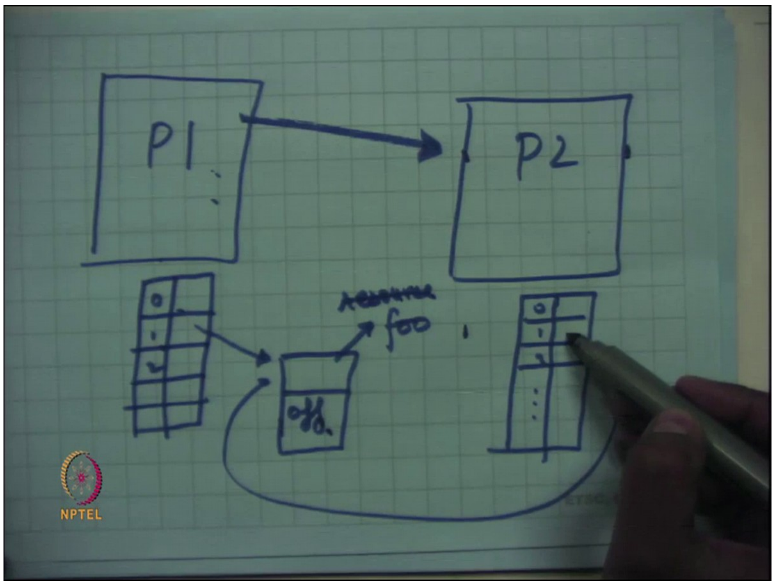
```
$ sh < foo > bar

foo:
    echo ; one
    echo . two
```

I could also do something like sh less than foo, greater than bar. What is this doing is; it is redirecting the standard input to foo and rewriting the standard output to bar and if you do it in pretty much the same way as we discussed earlier. You will close the standard input, open foo, close the standard output, open bar and execute sh .

If I wanted to execute a shell script; all I need to do is let us say this is my foo file; it will have let us say a few commands echo one and echo two. So, now foo can be a shell script and when I say shell is less than foo; the shell script gets read as input to my shell and that gives you shell script foo .

So, very composable here when I said echo one, I did not care about where the data is going I am basically saying I want to write it to standard output and the standard output was actually initialized much later and the invocation of the shell script foo . When I say shell sh is less than foo is greater than bar, it basically executes the shell script and prints one and two on the shell on the in the file bar.

(Refer Slide Time: 22:23)



So, let me just draw the file descriptor table once again. So, let us say there is a process P 1, this is another process P 2; there is a file descriptor table that is associated with the process P 1 and there is a file descriptor table associated with process P2 . The process P1 can only access this file descriptor table using system calls. So, I cannot a process P1 cannot just access this table directly; this table is sort of hidden from the process and the only way it can actually manipulate or read these tables is using read, write, open, close system calls.

A single entry in this file descriptor table points to a structure which says where is my resource and then it says what is the offset of that resource. So, the offset is maintained inside this structure. So, let us say the resource is a file foo and then there is an offset inside the structure which says what is the current pointer at which file is writing. So, when I call a system call like write on file descriptor 1, it basically just starts writing at the current offset off.

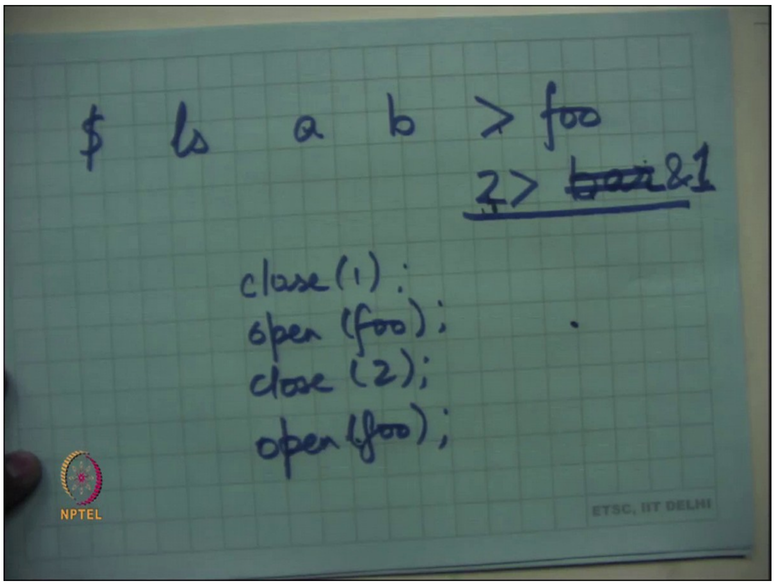
If there is another file that wants to write its again going to write off and so it is going to off. So, off is going to get implemented on every write system call, that is how two successive writes give the feeling of appending to the file; it goes to off. So, that is also part of the semantics of the write system call. So, the offset gets incremented on every system call it read or write.

Now, let us say and once and one thing I wanted to point out is; so when a process P 1 forks a process let us say fork P 2; then the file descriptor table gets copied and the pointers are shared . So, the same structures get shared and this allows you to do what is called file sharing . So, the child could be writing to the same file and the parent could be writing to the same file and their outputs would now get intermingle; in any arbitrary order depending on the execution order.

You could insert synchronization between your child and the process to decide the order in which you want to write to this file, but in any case this implements file sharing. There is another way of inter process communication ; So you a process forks another child process, the file descriptor table gets copied, the file gets shared whatever one process is writing is now visible to the other process for reading.

However, if process P2; let us say calls close on file descriptor 1; that has no effect on the file descriptor table of process P1 . Because the file descriptor tables themselves have been copied; it is just a pointer that have been shared here . So, if the process P 2 calls close; absolutely no effect on process P 1.

(Refer Slide Time: 26:04)



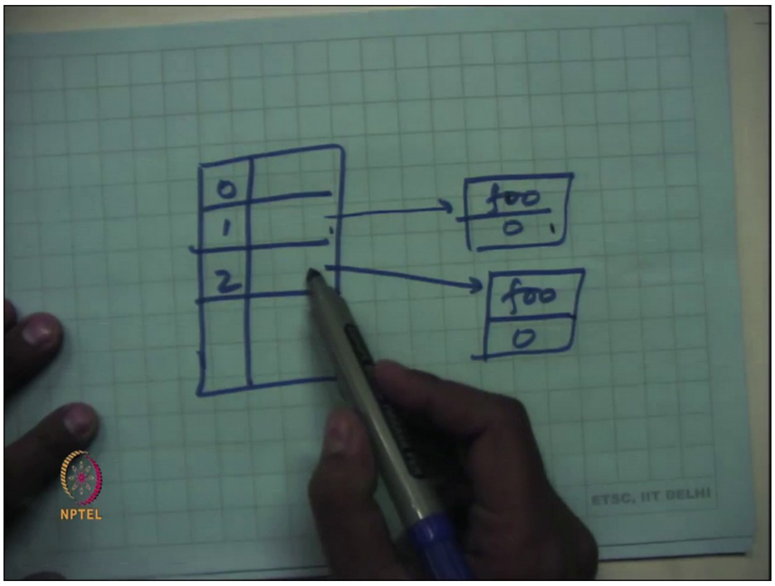
```
$ ls a b > foo
             2> ~~bar~~&1

close(1);
open(foo);
close(2);
open(foo);
```

The shell also allows you to do something like “ls a b > foo” and so this means redirect the standard output to file foo and “2 > bar”; which means redirect the standard error code to file bar that is it very easy and you can imagine how this can be implemented. But what it also allows you to do is it allows you to redirect standard error code to the standard output port.

So, basically saying redirect the standard error; whatever is written to the standard error to the same resource that is being pointed to by standard output. So, that is the syntax for that one shell is 2 greater than and 1 depends on different shells, but let us say this is syntax for redirecting standard output standard error .

So, how would one do this? How would a shell implement this? Well one neive way to implement this is to do the same thing which is close(1), open(foo), close(2), open(foo) ; things like the correct way of doing things ; let us see what happens.

(Refer Slide Time: 27:36)

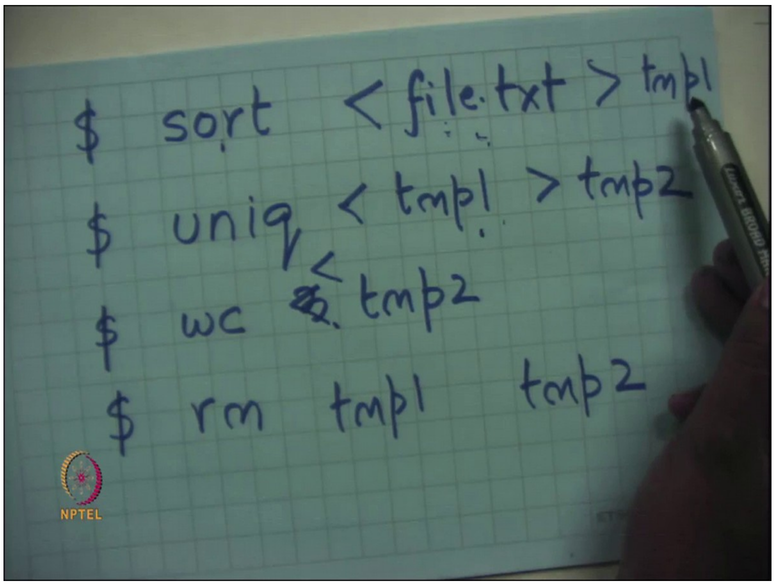


Here is a file descriptor table. So, when you call close(1) and open(foo); this point this gets initialized to foo and offset 0 and when you call open and open(foo); this gets initialized to foo and 0 . And now when the program was on standard output; it is going to write to file foo at a certain offset and then call write something to standard error, it is going to write foo and now what is going to happen is that they are going to overwrite each other .

What you wanted in this command was that they should get appended ; not overwritten . So, what do you need? It would have been nice in this picture instead of having something like this; I have something like this. So, they have a shared offset and each time they write either to standard output or standard error; the same offset gets incremented . The current set of system calls that we have discussed so far do not allow this kind of manipulation or the file table descriptor.

So, Unix has another system called; called dup ; it basically means that duplicate the pointer in the file descriptor table . So, the way I am going to implement this is I am going to say close(1), open(foo), close(2); instead of saying open(foo) want to call dup(1). So, that is going to have the desired effect of having a shared offset between standard output and standard input. Now, let us look at more interesting unique instructions.

(Refer Slide Time: 29:50)



```
$ sort < file.txt > tmp1
$ uniq < tmp1 > tmp2
$ wc < tmp2
$ rm tmp1 tmp2
```

So, let us say I wanted to; I have a file called file dot text and I and I have a utility which takes the file and sorts it . Let us say the it is an this is utility this is a program called sort just like there is a program called shell and there is a program called ls; this is a program called sort which takes input from its standard input and sorts it and prints the output to the standard output .

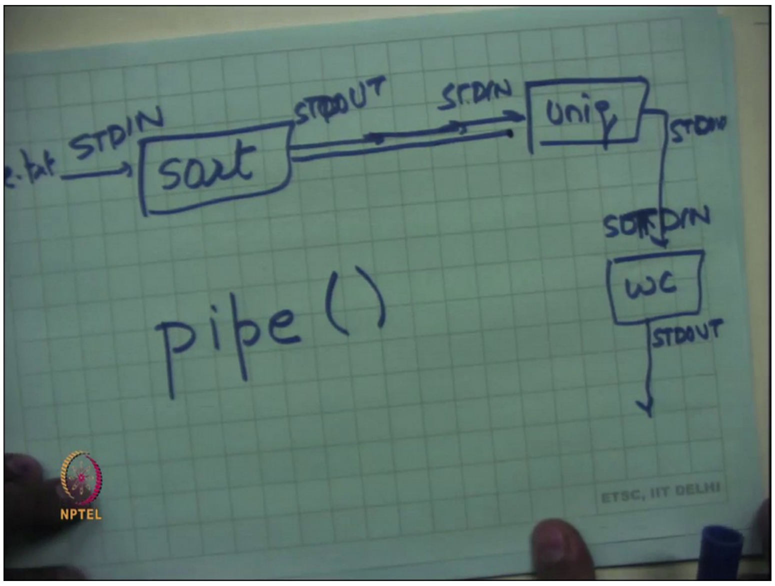
Let us say what I wanted to do was I want to first sort all the five all the strings in this file called file.txt. Then there is a program called unique which takes a sorted file and eliminates all duplicate entries from that; so just retains the unique entries in the file. So, one way to do this is I assort file dot text and redirect it to let us say some file temp 1, then call unique on temp 1 and redirect it to temp 2 . Then let us say I wanted to count the number of unique words in this file.

So, there is another command called wc with word count and I want to say a word count less than temp 2 or let us say I just say work let us say work count less than temp 2 and so this the combination of all these three commands is going to count the number of unique words in this file called file.txt. Sort the file find the unique elements and count the number of unique. Of course, I have created two extra files in this the process; so maybe the last thing I need to do is remove temp 1 and temp 2 .

This kind of a thing is also very common where the output of one program needs to be fed as input to another program . This is once again very very consonant with the concept of tools where there are multiple tools and you want to say I want to use this tool and then feed the output to this tool and then this tool and so on . If I am doing it in this way; let us say my file is hundreds of gigabytes or terabytes large. I need a lot of temporary space to be able to maintain this information number 1.

Number 2, I am writing extra things to the disk and writes to the disks are generally costly and so I am making things very inefficient. So, I am first writing the whole thing to the disk, I am reading the entire thing from the disk then writing another set of data to this and so on and this is all very expensive.

(Refer Slide Time: 32:45)



Ideally, what should have happened is that I should have had a way on my operating system to specify that here is a program sort, connect its standard input to file dot text and connect its standard output to the standard input of unique. And then connect the standard output of unique to the standard input of wc and connect the standard output of wc to whatever the standard output of the shell .

Because you basically saying just let say the shell was executing the console; you want the output on the console . So, if I was able to do this; if I was able to specify this then it would be very nice because I do not need these extra its temporary space and I do not need extra temporary time to create these space the space .

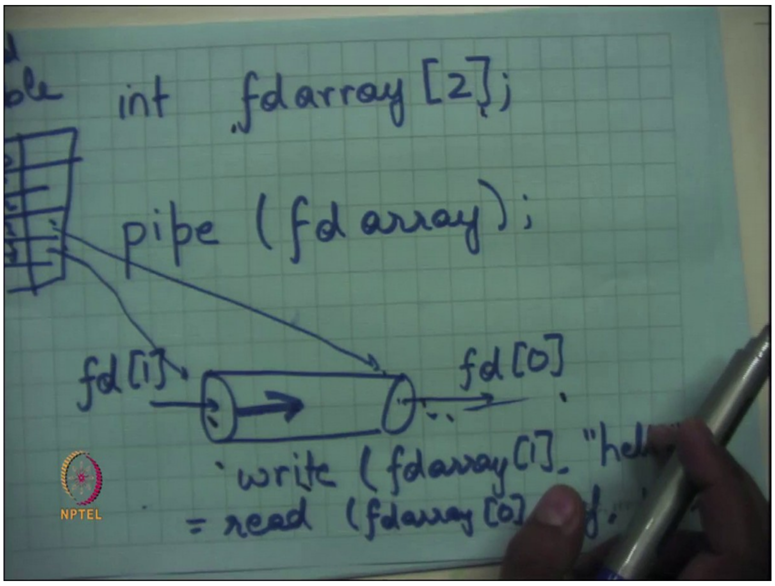
So, once again this requires manipulation of the file descriptor tables in certain way and once again I can know a process is not allowed to manipulate the file descriptor table in any way it likes. By the way why does not the process allowed to manipulate it file descriptor table in any way it likes?

For security reasons; you basically saying that these file descriptor tables are referring to shared resources; like the console, the file system, the devices and the OS needs to interpose on any request made to these resources. So, the OS will check on an open; whether this process is actually authorized to access this device.

For example, if this process is being run by the root user then it can access certain devices whereas, if it is not running as it root users cannot access certain devices. So, there is the ways I mean; so the OS needs to provide gates through which a process can enter; the process cannot just walk over my file descriptor table anyway it likes for securities. But now I need and then other OS is basically; the OS designer is basically choosing the write sort of system calls such that the processes are still able to do what they need to do must try.

So, I want to be able to do something like this and this; this connection between one file, the output file descriptor on one and the input file descriptor on another is facilitated by a system called pipe. Pipe was a very very new concept that Unix introduced and very successful since then ; so let us see what pipe does.

(Refer Slide Time: 35:55)



A pipe is a one way communication channel ; so here is an example shone in the above diagram. Let us say I create an array which is my fd array of size 2. So, the 2 fds in this array and I call pipe on fd array. So, this is the syntax of pipe as I am supposed to call the pipe system call on an array of integers of size 2 and what it is going to do is its going to create two file descriptors in my file descriptor table and put the output end of that file.

So, it is what is going to do is its going to create a pipe inside the OS. Once again this pipe is inside the OS and the process cannot just access it directly; it has to access it through gates which are system calls, it creates a pipe and it assigns two file descriptors to the two ends of the pipe .

So, for example it just let say there is the file descriptor table. what is going to do we just going to find the first two available file descriptors in this table; once again walking from the top. Assign the first one to the input end of this file and assign the second one to the output end of the file .

So, let us say the two file descriptors are available for 3 and 4. So, 3 is going to point to the input end of the file or to the depending on; so let us take this pipe as the entities. So, the output end of the file is the first one and the input end of the file is the second one .

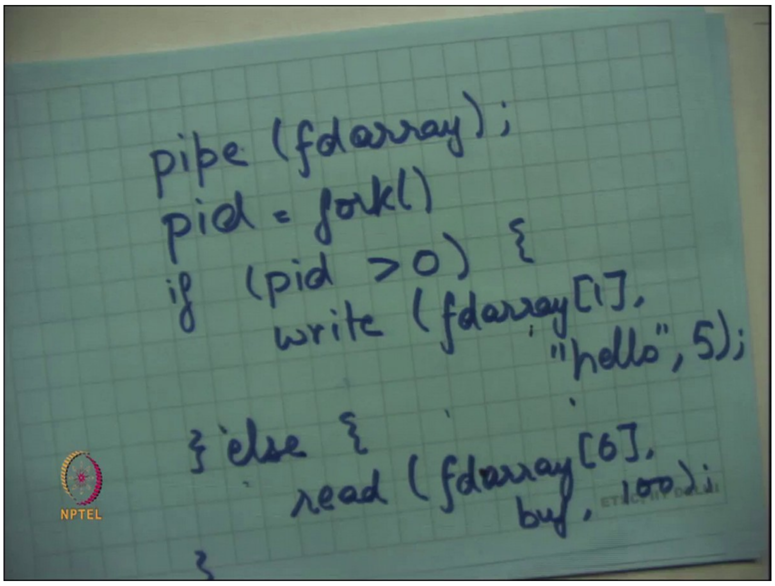
So, with that in mind if this program says pipe fd array and then it calls on fd array[1], let say says hello string of length 5 and then it says read fd array[0] in some buffer buf and let us say its maximum length is whatever 100, then what I am going to get is answer 5 and the string hello written inside buf.

So, what has really happened is when you call write; it wrote 5 bytes on this file descriptor which means 5 bytes were stuck into this pipe. And when you called read on this file descriptor; these 5 bytes will read of the output and out of 5; what you have done is to recall that file descriptors point to resources. So, far we have considered resources which are either external externally facing devices like console or sealcoat or they have files in the system which are again; So, there is a file system let us say. Here a file descriptor is pointing to an internal resource which is a pipe; that is created using the pipe system call . And both ends of this resource are internal ; in the case of a console the output end was in the OS. So, the input end was in the OS and output end was on the screen, but here the input end is also in the OS and the output end is also in the OS; that is one way of think about it .

So, when I write to the input end of this pipe in fd array 1 and then read from the output into this file I get what I wanted to I get what I wrote . So, this does not sound very interesting; I mean if I am going if I just write and then I read what is the point of doing this through a pipe, I could have just done it internally; I did not have to involve the OS in this doing then, but what this allows you to do is allows you to do inter process communication across processes.

So, as you can imagine if I call the pipe system call; I have initialized my file descriptor table in this way and then I call the fork system call, the file descriptor table gets copied to my child with the pointers remain the same; so they still share the same pipe. Now, the parent writes to the input end of the pipe and the child reads from the output end of the pipe; you have implemented inter process communication .

(Refer Slide Time: 41:00)

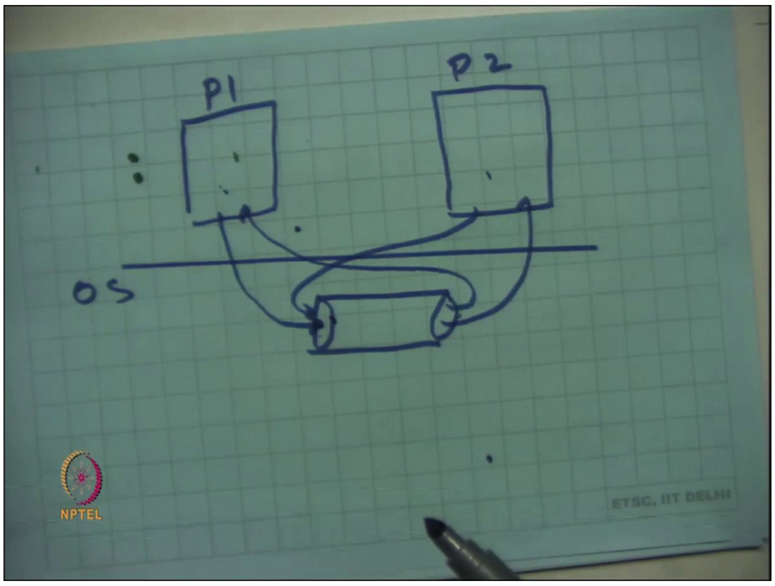


```
pipe (fdarray);
pid = fork()
if (pid > 0) {
    write (fdarray[1],
               "hello", 5);
} else {
    read (fdarray[0],
              buf, 100);
}
```

So, let us see how this works; I could call pipe fdarray, this is going to create a pipe . Then I am going to save fork(), this is going to duplicate the file descriptor table when; once a duplicate the file descriptor table the pointers get shared; so the pipe gets shared .

Then I can say if pid is greater than 0 which means I am the parent; I write to fdarray[1], hello . And else if I am the child, then read fdarray[0], buf and whatever 100 all in this case the parent writes to the pipe and the child reads from the pipe . I created a pipe, I forked after creating this pipe which means that the pipe now gets sheared.

(Refer Slide Time: 42:43)

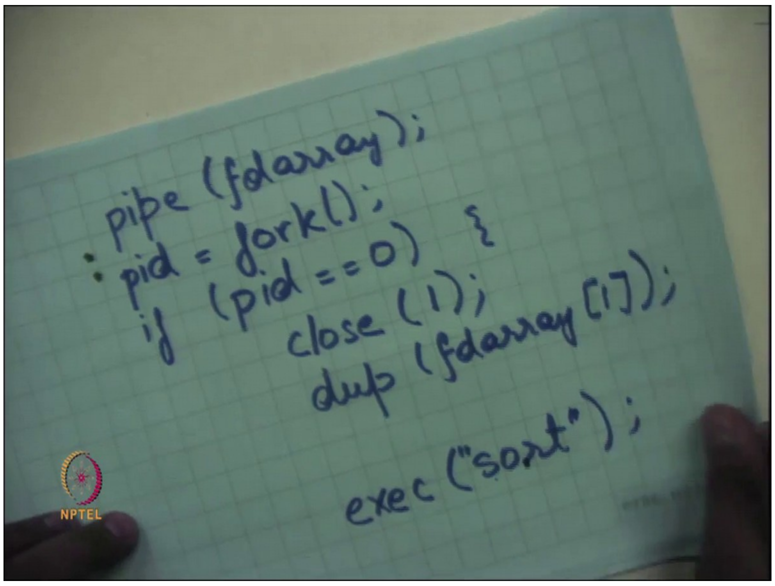


Let us just see what happens here is process P1, here is the OS, the process created a pipe. So, the pipe gets created inside the OS space which means it cannot be accessed directly by the process, but it can be access using read and write system calls.

Then the pipe has pointers which in which are the pipe which are represented by file descriptors. So, this is let us say fd array 0 and this the fdarray[1] and this is fdarray[0] . At this point the process P1 calls fork which means another process gets created P2, but the file descriptor table also gets copied which means that even this process as file descriptors which point to the same file; same pipe.

So, now if this process write to the input end of this pipe either P1 or P2 can read from this pipe; they call and also. One way to use this is to for P1 to call write on the write here and P2 to call read here and that way you can pass information between P1 and P2 . So, let us look at the example that we had earlier; I wanted to create pipes between sort, unique and wc programs and so I wanted a picture some which look like this file dot text sort standard output connected to standard input of this and so on.

(Refer Slide Time: 44:52)



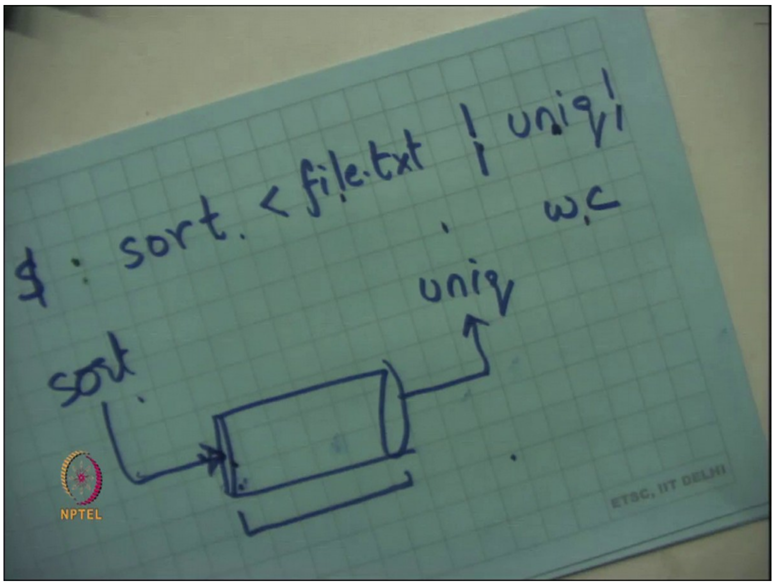
```
pipe (fdarray);
pid = fork();
if (pid == 0)  {
      close (1);
      dup (fdarray[1]);

      exec ("sort");
```

And so one way to do it is you call pipe on fd array, then you call fork and then you say if it is equal to 0 which means I am executing in the child context; then close the standard output, I want to connect the standard output of my child to the standard input of some other process . What I am going to do is I am going to close the current standard output of the child and I am going to call dup on fd array 1. What this going to do is; it is going to copy the file descriptor value in fdarray[1]; the point in fdarray[1]; two standard output.

So, now the standard output is connected to the input end of the pipe; fdarray[1] was input end of the pipe. So, when I did close 1 and the dup of fdarray[1]; I have connected the standard output of my current process to the input end of the pipe ; fdarray[1], . And then at some point I can say exec command the exec sort. I am hiding a lot of syntax just, let just understand (Refer Time: 46:40) when you execute sort; sort is going to write to its standard output and the standard output happens to be a pipe in this case .

Separately, I will connect the standard input of the other process that I am going to fork unique in and I am going to connect a standard input to the other end of the pipe and that way I am going to have a communication between sort and unique.

(Refer Slide Time: 47:04)



So, the syntax for this one shell is sort less than file dot text pipe and this is the pipe character on the keyboard I think it is and then say sort and then say unique and then again pipe and wc . This does exactly what we saw earlier which means its sort is going to take input from file dot text, pipe it to this pipe; pipe the standard output to unique. Unique is going to take the input from the pipe and pipe its standard output to wc; wc is going to take its input from the pipe and pipe this and print the standard output on the console.

And what I want you to think about after you go back is how we are going to, how the shell will implement some this syntax. So, this is the syntax that is provide the shell; its implemented by the shell using the file system call and I want you to write code to be able to do that . It is going to be something very similar to this, where you create a pipe, you fork and inside the child you manipulate the file descriptor so that it standard output now points to the pipe and then you exec the program that you wanted to .

So, when I created a pipe; there is the pipe sort is writing here and let us say unique is reading from here. what happens? So, now there is an interesting scheduling question that comes up; what happens if sort gets to execute a lot and unique has not even started . So, now sort is just producing a lot of data and unique has not gotten chance to run . So, now that brings to the question how fat is this pipe or how large is this pipe.

So, there are default values with the size of those pipes buns; so they and always there will be a producer to the in this pipe and there will be a consumer in this pipe. If the producers producing too much and the consumer has not even is not consuming at the same rate and eventually the pipe will get full; depending on the size parameter and the size is typically and if you tens of kbs; let us say.

So, once the producer gets full; the operating system can figure this out because the only way the program can produce is using the write system call and so the operating system sees the write system call it says where it is going its writing to this pipe it figures out of the pipe is full; it says let us block this process. When it blocks this process, it basically means this process will now not get a chance to run till the pipe has more space that makes things more efficient .

So, in a way now sort and unique programs can now dance together . So, sort produces something, get suspended; unique gets a chance to run, uniques consumes; pipe gets empty, unique gets suspended sort runs again sort unique sort uniques . If the buffer file buffer was very small you will always have this kinds of a scheduling patterns sort unique sort, unique, sort, unique if the buffer was larger then it can absorb more things.

So, they can happen any arbitrary order; it can be sort for unique sort for unique unique sort whatever . So, this way of doing things is; firstly, it is more space efficient. Secondly, more time efficient; you do not have to write anything with this and it also gives you an gives the operating system nice and insight into the scheduling the kind of scheduling that should happen for the system to execute at full throughput.

Student: Sir, does the operating system (Refer Time: 51:33).

So, it may or may not that is and that is just an optimization even if it does . So, the user can control the size of the pipe; there is a maximum limit that the size of the pipe can be; pipe on current operating systems are implemented in memory you never. So, on operating system does not typically use the disk to implement the pipe.

So, if its execute; if the pipe is being implemented inside memory, you are automatically bound by the size of the memory number one and actually much smaller than that because you want memory to be available further things too. So, pipes are typically much smaller than what you would have; what if you had implemented them in disks;

Student: (Refer Time: 52:19).

So what if the pipe is so small that even one write cannot complete properly; that is no problem. The semantics of write that till it writes entire data let us say this semantics (Refer Time: 52:36) write are till it tries to limit direct data it blocks . So, if you want to write 100 bytes and at 50 th byte it got blocked it just gets blocked and now the other guy the other process can should be somebody some reader should actually wake up if any and if it breaks up its going to consume.

And so the write has not yet returned and yet scheduling transfers are taking place . So, scheduling transfers are not necessarily demarcated by execution inside the application; scheduling transfers can happen in the middle of the execution of a system call.

Student: (Refer Time: 53:16).

Interesting. So, unique is giving its output to word count and the word count ; word count is supposed to compute the count of the entire file and if I just received half the word count; then what does it mean really a word count to ; word count cannot really work. So, the way word count program must have been written is that it is going to read the input, till it reaches end of input and then it is going to compute something on that input . So, till unique is producing something, word count is consuming something; word count will never produce anything till that has seen the end of input marker .

So, in this case unique; so sort is producing for unique and unique is producing for wc and these programs could be written either in either in online fashion which means I read few hundred bytes of your thousand bytes of a million bytes and then I can compute something on that or they can be written in a way that I want the entire input and only then I can compute something on that.

Student: (Refer Time: 54:32).

In their own applications space ; so they have allocated buffers in their own application space; they are going to put it in to their application space and now they are going to compute on it. Just like you write a normal program you; you read from a file you store it in an array and you compute something on the array; instead of reading from the file; in this case you reading from the file.

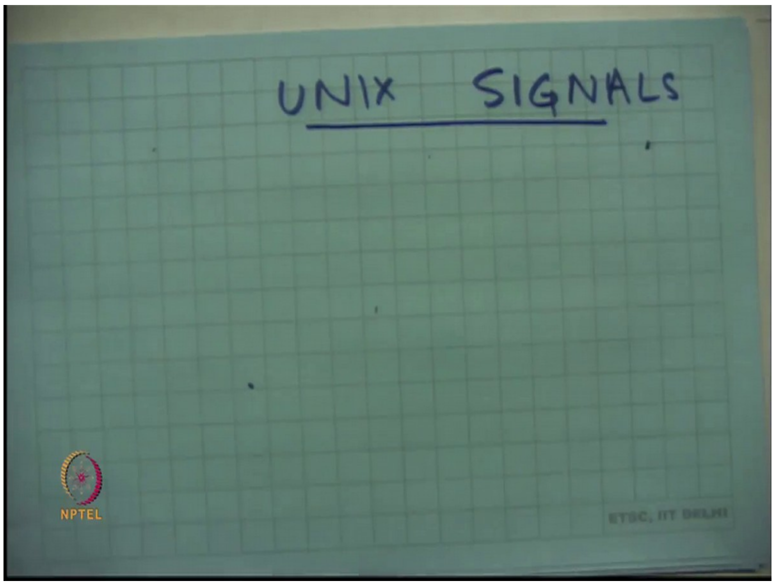
Student: (Refer Time: 54:57).

So, read will only return values that have ever been written to the pipe; it will never return any unnecessary garbage values and it will return the values in exactly the same order, in which they will return to the pipe. So, read will never return any wrong value really and if you say I want to read the next thousand bytes; till 1000 bytes are available let say the read blocks . Or it returns saying I have not; I have not been able to read I only been able to read 2 bytes and here the 2 bytes and now the it is; the to the programmer to say give me the next 998 byte .

How does the OS release the resources for the pipe? Interesting question; anybody? There is a system call that we discuss close . If you close both if you close the ends of the pipe if all the processes in the system, so a pipe is associated with the file descriptors of certain processes in the system . If you can figure out that all the ends of this pipe are closed so no process in the system actually points to this end of the pipe and the pipe is empty, then you might as well just completely release the resources.

Because if somebody calls read on that pipe you can just say that you can block it you can say nobody will write to it there is no writer to it. So, the pipe one end of the pipe is completely invisible to the system and so at that point you can release the resources for example, good the next. So, pipe is a very interesting Unix subtraction.

(Refer Slide Time: 57:07)



UNIX SIGNALS

The next subtraction I am going to talk about are Unix signals and now we are going to do this in the next lecture.