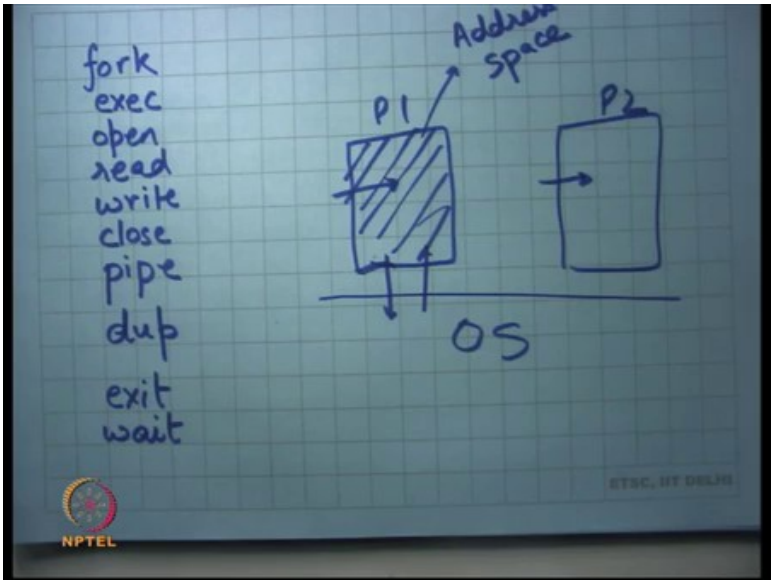
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 03
Threads, Address Spaces, Filesystem, Devices

So, welcome to Operating Systems lecture 3.

(Refer Slide Time: 00:28)



In the previous lectures we have loed at a Unix and its interfaces, in particular we loed at the system calls fork, exec. Fork was a system call which copies a process and makes a new process out of it, exec overwrites the current process with an executable and started running.

Open, read, close, operate on files or other resources and use file descriptors as handles. Pipe is a way of creating an inter process communication channel between two processes, dup allows you to duplicate a file descriptor inside the file descriptor table. Exit is a way of telling the operating system that the process has finished and wait is a way of telling the operating system, that I want one of my child processes to finish. And, I want to get its return value or the exit code or the exiting process got it.

There is operating system and various processes like P1 and P2 running on the top of it. What is a process? Processes you can think of it as a running program, a program which

is an action which has state associated, running live state associated with it. The abstraction of a process says that the entire space, where it is running in main memory is entirely private to it.

So, if I allocate something inside the process it remains completely private to me, if I write something to in this space it remains completely private to me and nobody else can touch it. So, this space is also called an address space. So, this is an address space, it is called an address space because if I say I want to access address 1 million then the address 1 million here will point different places in p1 and p2.

So, the same address in different address spaces points to different locations and an address spaces is also an abstraction that is implemented by the operating systems. So, process also means that it will have its own address space. The nice thing about this kind of isolation between processes is that different processes can run completely independently and they do not need to be worried about each other and, worried about what they are going to access and whether I should protect myself with another process or not. So, as an example your web browser does not need to worry about a shell or your compiler other thing. They can all sort of execute independently because they are executing in completely independent addresses spaces.

Whenever they need something which requires access to shared resources and the shared resources could be devices, they could be files etc; they use things like system calls or even memory. So, if they want more memory anything; so, clearly you have one physical system that is being shared by all these processes.

If I want to get access to these processes these resources the process needs to ask the OS for it and the OS is going to decide whether this request is legal or not and appropriately arbitrate these resources among these processes. And, also ensure that concurrent accesses to these resources from these processes are protected.

And so, fork exec open are is one way of doing these things, fork is a way of creating a new process. So, process the list of processes is also in some sense a resource and if you want to create a new process you ask the OS can I want to create a new process. The OS checks if it has enough resources to actually support a new process for example, does it have enough memory to support a new process.

And, if it does it will create a new process and it will return the ID of the new process, if not then it will return an error code which will be a negative number. And, similarly if you open a resource like a file whether you have permissions to open that file or not. So, in some sense the OS is acting as an arbitrator between the process which is untrusted and the resource.

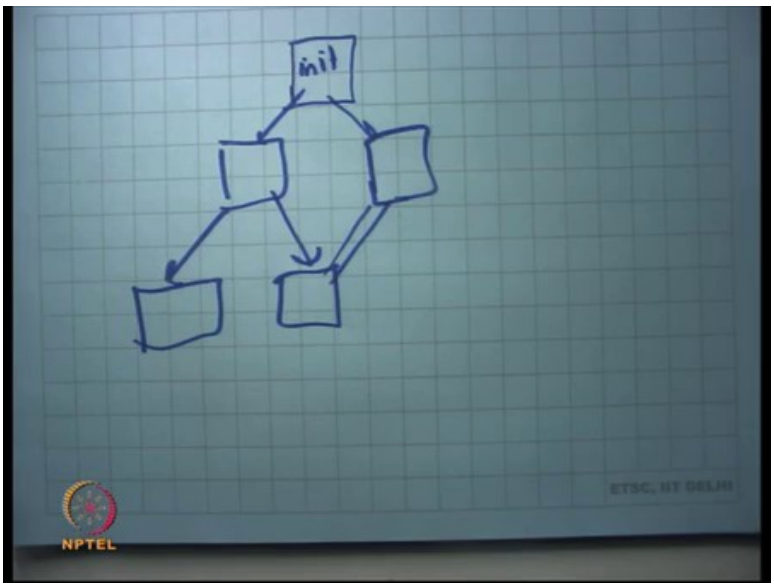
So, process can be thought of as an untrusted entity, it is a program that is running, you do not really trust it, in the sense that you do not trust it to the program could be greedy, it may try to take too many resources, it could be malicious, it may be trying to crash the system or it may be just buggy; in either case you do not want a buggy program to crash your entire system. So, it is a job of the operating system to ensure that all your resources remain safe even in presence of these untrusted processes. And so when you design these interfaces you also take one of the important things that you have to worry about is security. , you do not want untrusted processes to be able to take control of a system in unexpected ways.

And, another thing you have to worry about is performance and we saw some examples of a how different interfaces have different performance trade-offs for example, we looked at create process was this fork and exec and other things. As we go along the course it will become clear why this interface is reasonably performance and that is why it's popular. So, both performance and security are quite important when you are designing such an interface.

Now, the process P1 can communicate to the OS, but often a process needs to communicate another process. And, we saw some examples we saw one example last time we have one process wants to wants to compute something on its input. And, then pipe its output to another process which wants to compute something on its input and then and so on.

So, a chain of processes with pipes connected in the middle and this is a typical example of a situation which requires inter process communication and, pipe is a system call to be able to do this inter process communication efficiently al and then we have also looked at exit and wait.

(Refer Slide Time: 06:45)



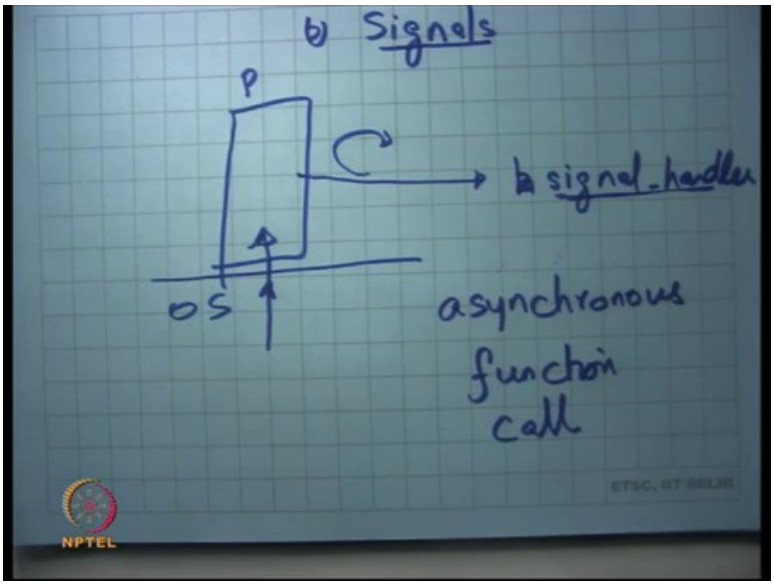
so, the processes is in a system are often represented by a hierarchy which is the hierarchy in which they were created. So, for example, when you boot your system the operating system will typically create one process which is often called the init process. And this init process is going to fork more processes and more processes and these will spawn more processes in turn. And, also if needed the init process will create pipes between these processes and so, the system looks like a clear processes with pipes connected between them or inter process communication channels between them in gener.

So, for example, when I when the system boots up it will spawn let us say a shell, it will spawn the x server, it will spawn let us say some default applications. It will tell the default applications that your default terminal is this shell or it will tell the default applications that your default terminal is on the x server and it will connect them appropriately. So, these things can run simultaneously and relatively independently.

So, you have many programs that are running independently and yet they can communicate with each other to do the things that you want them to do. I will point out here that pipes are one way of doing inter process communication and there are other ways of doing inter process communication which are pretty much similar in their nature where you make system calls to be able to write things to that you create a channel.

And, then you make system calls to be able to write to that channel and then you make system calls to be able to read from the channel.

(Refer Slide Time: 08:27)



So, that is a review of what we have done so far and today I am going to talk about another abstraction which are called signals. Often when a process is running you want to interrupt it and you want to say I want to do something. So, process is conceived as a sequential flow of instructions. So, one instruction after another is running that is the normal flow of execution of a process. But, let us say I want to bring to the attention of a process a certain event for example, a key was pressed. So, that is done using what is called signals.

So, a process is called a signal with a certain signal and the semantics of a signal is that it amounts to an asynchronous function call. So, let us say this is a process which is running, this thing means that its running and now the OS wants to raises the sign. What this means is, the semantics of a signal is that it will interrupt the current execution of the process and make a function call to the signal handler. So, whatever it was doing it suspends that in some sense and makes a function call to this special function which has been which has been registered previously and makes a call to that function. And that special function is called a signal handler, that signal handler also executes in the address space of that process.

So, the signal handler executes in the address space of the process and can touch all the memory that the process can touch or see read and write all the memory that the process can read. when the signal handler returns the execution continues from where it was interrupted. So, that is the semantics of a signal, in some sense it is like an asynchronous function call.

Student: (Refer Time: 10:31).

Yes.

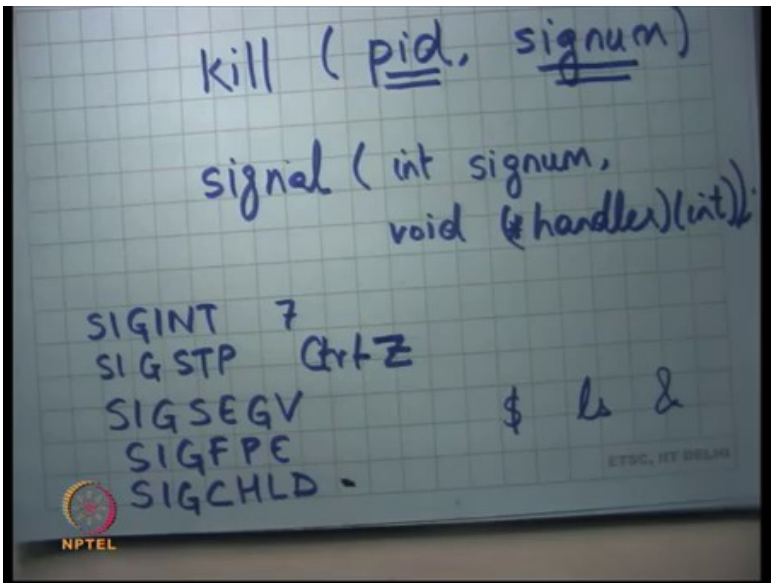
Student: Sir, you said that signal handler executes in the address space of the process. So, can it happen that the signal handler and the process that is currently being execute are totally different, they are not related. So, can that happen?

So, by not related you mean?

Student: We registered signal for process 1 and by scheduling policy suppose OS is running process 2 and then this signal came up.

So, signal handlers are private to a process. So, if process 1 registers a signal handler then while process 2 is running and a signal occurs, process 1 signal handler will not get executed; process 2 will have its own signal handler. So, every process will have its own signal handler and those signal handlers will execute in the context of its own process. So, this can be thought of as an asynchronous function call, it's basically like saying that whatever you were doing just insert a function call after that. So, you call that function that function when it returns, it just returns exactly where are you call it and you can start executing the next instruction. So, the reason this kind of an abstraction is needed is to be able to handle rare events, events which occur which are sort of events to allow a process to be able to we can expose an event driven interface, like press of a key , arrival of a network packet or something like that. So, you want some kind of an event driven abstractions for a process to be able to do more than just sequential execution.

(Refer Slide Time: 12:21)



So, there are two system calls on Linux: one is called kill and another is called signal which support this Unix signal abstraction. Both of these are system calls and signal takes two arguments; the first is a signal number and the second is a function pointer to its handler. So, process can actually register handlers for signals. There are a set of standard signals that are pre-documented which are part of the OS semantics.

For example, there is a sig interrupt SIGINT, the interrupt signal which is triggered when you press a control C while the process is executing. So, if you press a control C what happens is the operating system figures out that control C has been pressed on the standard output of this process and so, it generates a SIGINT for that process.

What that causes is that, it causes an asynchronous function call of the SIGINT handler, the default SIGINT handler will simply exit, will simply called the exit system call and the process stops . So, that is why that is why you see that most of the times when you press control C the process just exits.

And, operating a process may choose to overwrite the default signal handler, the default SIGINT handler with its own handler for example, I just want to count the number of times control C was pressed on this while this process was executing. So, it can do that for example, or it may say whenever user press control C then free this data structure or start a fresh, it all these are possibilities.

Student: Sir, what does a process acquire default SIGINT handler?

That is just a part of the semantics. So, the Unix the operating system will say this will be our default signal handler and it will map it in the space address space of that process .

Student: Signals response is used to register the signal handler.

Yes, signal the signal system call is used to register a signal handler with a particular signal number. And, SIGINT has a particular number let us say; I do not know what it is, but let us say its 7 hypothetically. So, SIGINT stands for 7 and so with the signal interrupt number 7 the handler is. Similarly, there is sig stop which basically suspends the process and control Z, delivers the sig stop signal to the process.

So, I said the signal system call allows you to overwrite the signal handler, the default signal handler or whatever has been previously registered with a new handler. Now, the question is can I overwrite all signals? For example, there is a signal called sigstp which is which you can invoke by typing control C on control Z on the standard output. And, you and if the process is actually allowed to overwrite this signal then there is no a user will completely lose control of what the process can do.

So, there are certain signals that the operating system will allow you to overwrite and there is certain signals that you it will not allow you to overwrite. And so, six stop happens to be one of those signals that cannot be overwritten. So, clearly the signal system called as a process called the signal system call, the OS can decide whether it wants to allow this overwriting or it does not want to allow this overwriting. And, there are rules when it will allow and it will not allow.

Student: Sir, the integer argument for the handler function sir, that is the signal number itself.

Yes.

Student: Sir so, if you are passing a given signal number and we are passing a function for all possible signals when we call the signal (Refer Time: 16:24).

No. So, that handler is associated with that particular signal number.

Student: Sir, but it takes an integer arguments so.

.

Student: This is only for a just a single signal, then it need not take an argument.

. So, this is a simplification often what people do is use the same function to handle multiple signals. And so, this argument of this function allows you to disambiguate what was the signals and you may want to take different actions depending on what you do; so, it just a simplification .

So, the signal handler if you notice takes an integer argument which basically says what was the signal number that was generated and so, that allows you to have the same function pointer for multiple signals. So, the same handler for multiple signals and .

Then there is another signal which I am sure a lot of you have seen it is SIG SEGV, this is segmentation fault. So, even the segmentation fault is treated as a signal. So, what happens is if the process does something illegal which means that touches the memory location that it's supposed to not touch, the operating system will figure that out in an efficient way and we are going to see how it figures that out in an efficient way.

But let us say it is able to figure this out that the process is trying to access and address which it does its not supposed to access, then what it will do is it will generate a segmentation SIGSEGV for the process . And, now the SIGSEGV handler is going to get called and so, the default SIGSEGV handlers is going to abort the process .

And, then similarly there is a sig floating point exception for example, if you do a divide by 0 you will divide by 0 you are going to get a SIGFPE and similar things as SIGSEG V are going to happen. There is another signal called SIGCHILD, this is a signal which is generated if one of the child processes of the process of the current process exits.

So, let us say I started a child process and I wanted it to run concurrently; I did not want to wait on it, I did not want to block. So, I wanted for example, I start a browser window and then I start an; I start a new window from inside the browser window and I want both windows to exist simultaneously.

And, now one of the windows is closed which means that calls exit and so, you may want that the parent process be notified that the child has exited for it to maintain certain

data structures about what are all my outstanding children for example, or anything else I did not want to do.

So, SIGCHILD signal is a way of the process or the parent process to know that its one of its child children has exited and now it can check what has happened really. So, for example, in the shell example when we type what happened was ls was spawned as a new process. And, the shell implementation we saw so, far used to wait on that process to exit. If instead I did ls ampersand, I do not know how many of you have used this syntax of shell. It basically means that you spawn a new process, but you do not wait for it.

So, you also let the shell continue execution and let ls continue as a separate process, ls may not be such a natural example, but think of it as a browser for example . So, browser can continue on the shell and you want to type more commands on a shell for example. But, what you also want to do is that when the child exits you want to know what it returns status was.

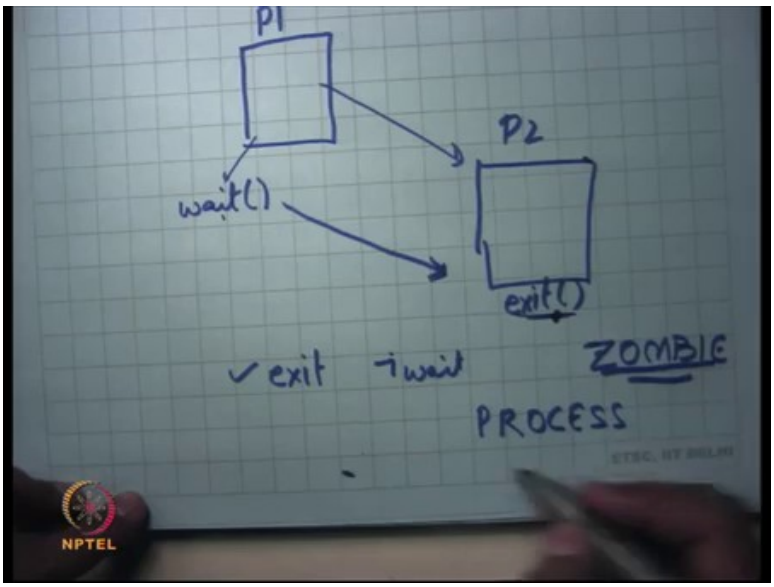
So, that is how SIGCHILD can help and finally, this kill system call semantics of the kill system call our that you can actually ask the OS to send a signal to another process. So, you can say that send the signal signum to this process pid .

So, one process can actually send a signal to another process by simply saying that I want to send a signal to this pid, this signum. Of course, there are rules I cannot send a signal to a process running by another run by another user, I can only send certain types of signals and so on . So, with those rules the skill system call allows you to do these things.

So, I do not know how many of you have used this program called kill which its the program goes rogue you just say kill dash 9 the program, that program is just calling this kill system call to and with the pid of that program to basically.

And, the signal that sending it is from SIGINT or sig kill or something sig kill and 9 stands for sig kill. So, that is an introduction to Unix signals, it allows you to handle interrupt based or signal based computation which is relatively rarer and which requires event driven handling to be doing efficiently .

(Refer Slide Time: 21:57)



Another interesting thing about processes is that let us say I am a process P 1 and it called one other process P2 and the process P2 finally, calls exit . And, while the process P2 has called exit the question is should the operating system delete all state which corresponds to P2 or should it hold on to the state for some more time? And, the reason I ask this question is because P 1 may actually call wait sometime later and wait is supposed to return the exit code of the process P2.

So, if exit of the child was called much before the parent called wait then its the responsibility of the operating system to preserve the exit code of the child. So, that when the parent calls wait it can return the value. So, in some sense a process is not cleaned up just after exit, a process is truly cleaned up after its parent calls wait on it.

Till the parent does not call wait on its child process, there is some state of that process that remains in the operating system. And, now a process which has called exit, but its parent has not called wait it is called a zombie. So, not wait, but exit it is called a zombie process.

Now, one easy way to remember this is it is like saying that somebody has died, but his “aatma”(english meaning is soul) is remaining because nobody is called wait on the [FL] or [FL] has not been done, it like we talk about it. So, that is a so, its zombie process the return code of the zombie process is actually just lingering around in the orient system and this kind of bug is actually very common.

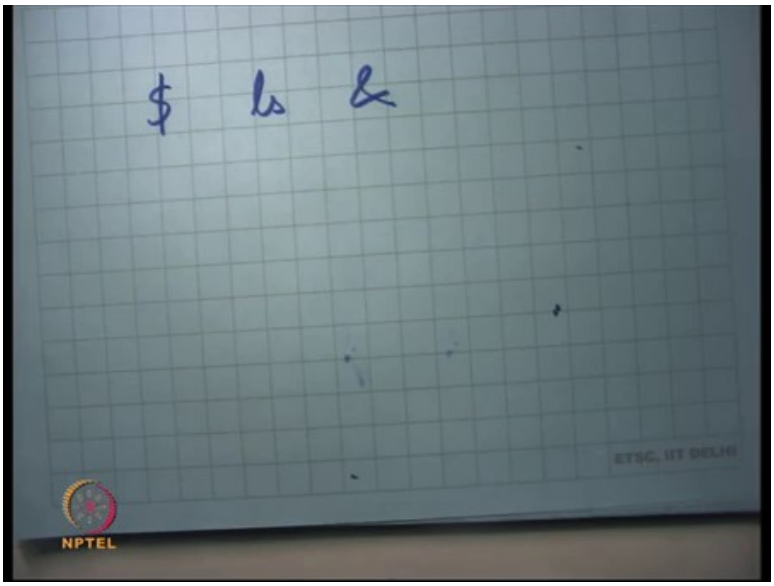
Because, what happens is that the parent the programmer forgets to call wait on its child and these kind of this is called a process leak or so, it is called a leak basically. It is a leak because you created some process some number of processes and you forgot to call wait on it. And so, these processes keep occupying some space in the operating systems structures which store its exit code and they will remain forever.

And, if it is a long running system then you can imagine that over months you are eventually going to run out of memory space, storing these things has a question.

Student: Sir, in the excessive bo it says that you wait call returns the pid of the exit child which was died.

So, x v 6 and Unix are slightly different number 1, the syntax of wait may be different, but the return code is definitely returned in the wait system call . So, I believe there is a pointer that the wait system call takes which basically gets stuffed with the return code. So, there are two ways; so, there is a return value of the wait system call and then there is a pointer that it takes where the operating system is supposed to stuff the return value. So, in any case the return value needs to be preserved till the wait system call is call.

(Refer Slide Time: 25:48)



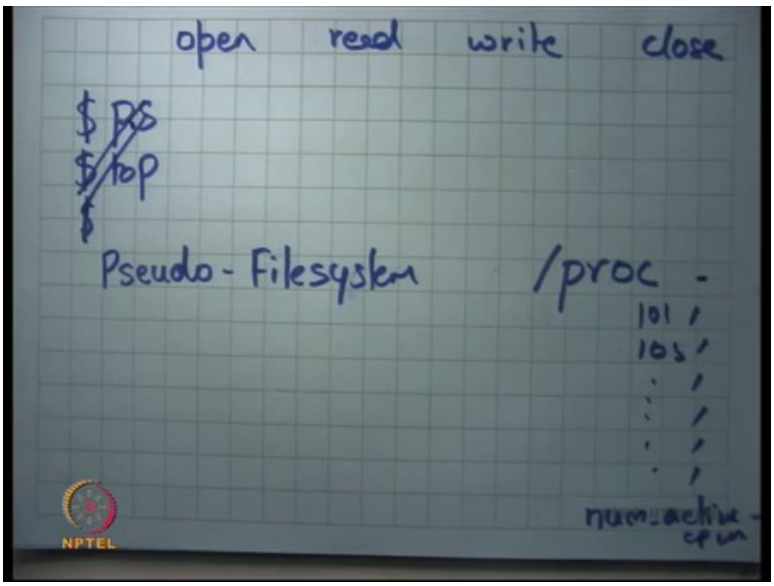
So, for example, on the shell if I say ls and if I implemented my shell by saying that I am not going to call wait on the child process, I just spawn the new process and I forget

about it and then I take the next command. What is going to happen is that will finish it will call exit, but it will remain in the system for ever because it will remain a zombie.

What is needed in this case is that the shell should, one way to do it is the shell code keep checking periodically whether all the children that I have spawned so far has anybody of them exited. Question is how often should I check? Should I check every 1 second? Should I check every 1 minute, 1 hour, 1 day? No, that is completely configurable, the other way to do it is to use signals . So, signals allow you to sort of more elegant way of doing things.

When you say that if a child exits, a SIG CHILD gets generated in the shell, the SIG CHILD handler checks all my checks which of my children have exited. And, if somebody has exited it calls wait on it, collects the exit status, does whatever it wants to do with exit status; perhaps it just wants to ignore the exit status whatever carries on.

(Refer Slide Time: 27:11)



Next, I want to talk about open, read, write, close and the power of these system calls. So, so far we have looked at open read write close in the context of a files and in the context of a devices and things like that. But, actually modern operating system use this system calls to do many more things.

For example, if I wanted to find out what is the number of processes in the system or if I wanted to find out which process has how much memory allocated and in which address

spaces . So, you can imagine that in one application typically may be interested in huge amount of information from the operating system.

For example, if you have ever used this command called PS it basically lists all the processes and what are their process IDs, what are the dependencies with other processes and how much memory they are consuming. So, there are other programs like top which tell how much CPU they have used, how much memory they have used etc.

So, a process may be interested in lot of information from the operating system, also a process may want to configure the operating system at run time in several ways. For example, then operating system can provide the functionality that you can on the fly change the number of running CPUs in the system. For example: you want to do some kind of energy saving computation and you want to say I only want the number the processor only I have 16 processors in this machine.

But, I only want the processes that I am using to be on everything, else can be off and you want an application to be able to control that let us say. So, an application to be able to say oh just switch of 10 process processors and I just want the remaining ones do work. And so, the you can imagine there is a plethora of different functionalities that an operating system is burdened with to provide to the user.

And, another system called we have seen so far really solve any of these. We have been talking at a very macro levels and all these different things are much more micro level. And, the question is how many system calls should have an operating system provide to be able to get this kind of functionality?

Seems very daunting. The way this is done in on Linux for example, is using a pseudo file system called /proc. So, what this means is there is a the /proc lives in your regular file system prefixed with the slash character, but it is not a real file system, it is a pseudo file system. /proc will have subdirectories which will be all the process IDs for example, /proc /101 103 and so on. So, these are all the process IDs and these are all sub directories. And, what an operating, what an application needs to do to be able to for example, if I were to implement PS, all I need to do is open /proc and read its contents. So, the open and read system calls can be overloaded to be able to do any information gathering from the OS.

So, these all the information that the OS wants to wishes to expose to the to its applications can be made part of this pseudo file system. And, an application just needs to use the regular open read write closed system calls to be able to read those system call, read that information.

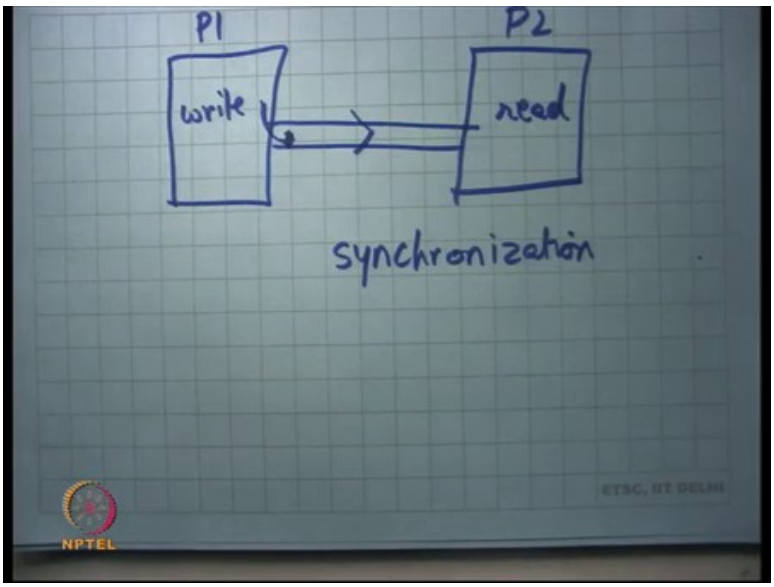
Of course the operating system can still interpose at an open call to see whether an application is indeed authorized to read this information. For example, I should for example, some of the simple rules are I should only be able to read information about my own processes and not allowed any other users processes and so on.

So, once again those kind of access controls can be made can be enforced at the open system call for example, or the read system call for that matter. Similarly, if I wanted to change the number of active CPUs that is also possible the let us say the slash proc has this file called num CPUs num active CPUs.

And, when a process says write to num active CPUs internally the operating system is going to trigger the procedure to actually switch off the other 16 minus 2 CPUs for example. So, you have overloaded the file system abstractions to do other things which are which involve getting and setting values in the OS itself. So, as an example I mean just to put things in perspective an operating system a full fledged operating system like Linux would have roughly 300 system calls al, that is still a lot.

But, it is still not in thousands or millions. So, far we have looked at processes how processes have private address spaces, how processes have abstractions to access resources that are provided by the operating system. These resources could be hardware resources, files or its own data structures for that matter. And, we have also looked at how different processes do inter process communication using pipes.

(Refer Slide Time: 33:30)



So, let us take an example of two processes which are trying to do inter process communication P1 and P2 and let us say there is a pipe. So, let us say this is the producer and this is the consumer. So, he is going to call on this pipe and he is going to call read on this pipe.

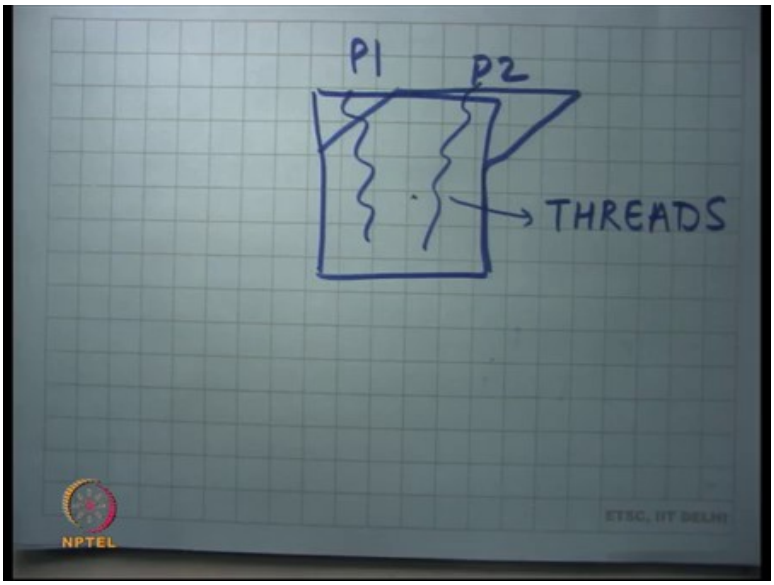
This pipe also acts as a synchronization mechanism because, if P2 calls read and the pipe is empty it basically means at P1 needs to run some more and call some more writes before P 2 can execute. So, in some sense they both synchronize with each other.

Synchronization means that they sort of communicate with each other and restrict their behaviour based on the behaviour of the other, that is one we are doing synchronization. This way of inter process communication is slightly expensive in the sense that to be able to write you need to make a system call and we are going to see later what is the cost of a system call.

So, a system call means you have to tell you have to make a function call inside the OS which means the OS has to do a few things, get your arguments, process them. Then the other the receiver has to call and make another system call read and those arguments need to be given to the into the address space of the other process. So, there is some amount of overhead, there is a copy that is involved from the address space of the first process to the operating system kernel.

And, the and then a copy from the operating system kernel to the address space of the second process and these are relatively expensive operations.

(Refer Slide Time: 35:11)



On the other hand if it were possible for two different processes to share the address space, you could have done this communication much cheaper. For example, if I want to send a data or message from P 1 to P 2 and we are sharing memory; all I need to do is set a byte in my shared memory and the other one is going to get that byte .

So, read and write two shared memory, the kernel is completely out of the picture in this case. And, you can imagine write in this case is just a memory write and read in this case is just a memory read.

And, both are significantly cheaper or more efficient than doing write and read system calls in a pipe. So, this now this scenario where two different processes shared the address space, these entities which these are the sort of different entities that are existing simultaneously, but they are sharing the address space are also called threads .

So, a thread is an execution flow; so, a thread governs an execution flow and two separate threads can have different execution flows. A process is a thread plus an address space. So, a process basically means it’s an execution flow with its own private address space. As opposed to a as opposed to two threads within the same process sharing the same address space.

What are some advantages of using processes and threads? Well we have already seen the advantages of processes, you could not have different processes running different programs which are completely independent of each other. One process could be waiting on could be running on the CPU another process could be waiting for the disk to get some data yet another process could be waiting on the network. And so, that way you have full utilization of your hardware resources.

So, for example, I have three processes, a browser, a compiler and a shell, the shell maybe just waiting for the keyboard input which means it is just blocked, its not it does not need to run on the CPU until user presses a key; the compiler may be actually running on the CPU and so, it is using the CPU, its keeping the CPU busy. So, while the first process is also running it's actually the OS has been able to multiplex these two processes in a way such that CPU is use in the most efficient way.

And, yet let us say the browser is waiting on the network card to receive the next packet and so on. If I did not have these this process abstraction being able to implement this, this multiplexing would have been much harder . Because, I would have had to have one program that understands that this part of the program is now waiting on the network and this part of the program is now execute on the CPU.

When you split into processes the and expose system calls to the processes the operating system has full visibility that this process is actually looking for the network. And, this process is actually executable CPU and that is where you have more utilization.

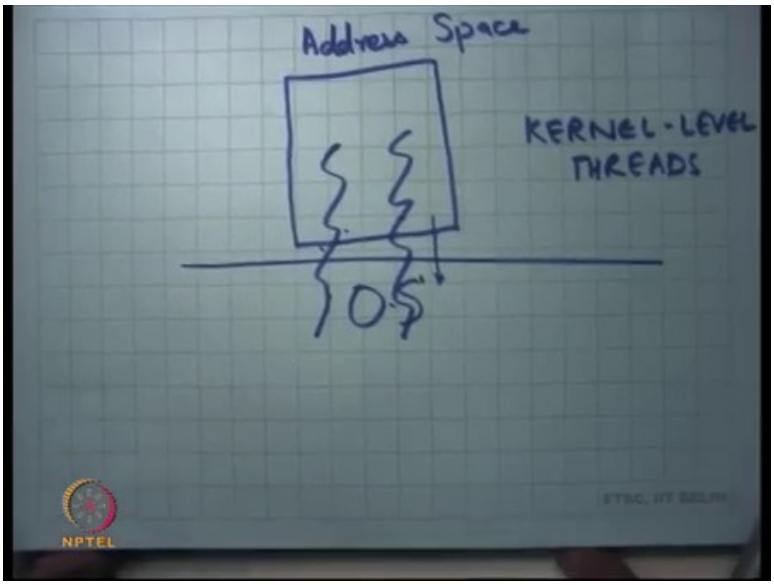
Further, when you talk about multi-processor systems the only way you can use other processors if you have multiple threads of control. If you just have one thread of control that is running then the other CPU will never be able to run. So, you need multiple threads of control to be able to actually utilize different processes, processors in your system. So, you need multiple processor processes to be able to actually keep multiple processors busy.

Same thing with threads; so, threads give you the same kind of advantages. You get more concurrency, you get more utilization out of your system. One thread could be waiting on the network, yet another thread could be executing on CPU 1 and yet another thread could be executed on CPU 2, all simultaneously to give you the maximum system

throughput. The advantage of threads over processes is that because they share address space they can have very fast inter process communication.

The disadvantage of threads over processes is that because they share address space there is no isolation. So, whatever one thread does the other thread is not protected from it. So, in other words two threads need to trust each other; what is more they need to be designed each such that they should be know they should know the existence of the other.

(Refer Slide Time: 40:30)



For example: the fire for the browser and the compiler do not need to know about each other because they live in separate address spaces. But, if you made them threads then they will have to know about each other and do appropriate safeguards for doing proper protection.

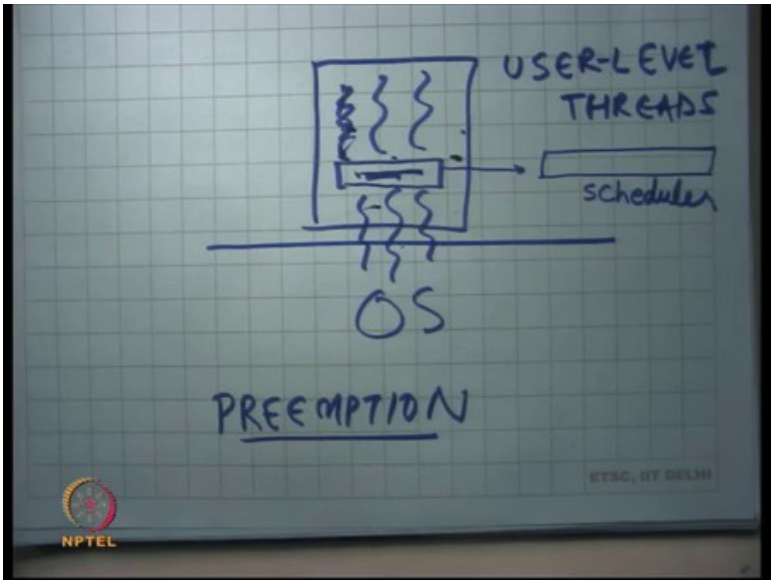
So, how does one create threads? Let us say here is a process and the instead of calling it a process I am going to call it an address space. So, here is an address space, we saw that this fork system call was actually spawning a new address space and also creating a new thread of control.

What we now want from the OS is to be able to create a new thread of control without spawning a new address space. And, the only way you can one way you can do it is to have another system call which creates a new thread. So, you could have multiple

threads within the same process, within the same address space and you would need to have some system calls on your OS to be able to do that .

If you do something like this then the operating system is aware that these are independent, independent threads of control. And, one of them can execute on one CPU and another one can execute on this another CPU provided the same address space gets mapped on both CPUs, the same memory gets mapped on both CPUs.

(Refer Slide Time: 42:00)



Another way of doing threads or of implementing threads could be that the OS has no idea about multiple threads. So, what you do is you tell the OS that I am only one process, but internally you have some kind of a system which allows you to emulate multiple threads of control al; there is a question.

Student: Sir, in this if I want to exit one of the processes and I cannot actually because, OS is that there are actually two processes running in the same addresses (Refer Time: 42:32).

So, in this scenario you are saying if I if one thread wants to exit then it will not be able it will cause everybody else to exit also, that is what you are saying yes true. So, so just one so, what I am trying to show is that the two ways to implement threads: one is to tell the OS to create multiple threads of control within the same address space in which case

threads are a first class entity. And, the OS knows about them and as OS is scheduling them across the multiple resources that you have in the hardware.

Another way of implementing threads is to not tell the OS about it, but to be able to do it internally. And, this is interesting because I am going to tell you next about how this can be done using the abstractions that you have studied so far. As you can imagine the OS has no idea that there are actually multiple threads being emulated within the process. And so, if one thread calls exit it basically means the entire process gets wiped out.

Further, if one of these threads called the system called read and let us say the read was on its doing a read to the disk and so, the all the threads will now get blocked because one thread calls read, its as though the entire process called read . And so, there is no real concurrency and so, the hardware resources are not being used to its fullest. Similarly, if one thread calls and read on the network then the entire process gets blocked and so on.

Student: If the abstraction is kind of one way that abstraction could have been used for some system false also, that read or exit could have been blocked and modified.

. So now, the question so one way to do this is you wrap the exit system call into your own system call which is called thread exit. And so, the thread exit will basically just be stopped within the thread and that is going to just free up data, all data structures related to this thread and the other two threads can still remain .

So, that handles things like exit, but what if I wanted to do a read from the network that still needs to go through OS because, no abstraction provides you a way of doing read directly so far. I mean the abstraction we have discussed so far do not allow a process to do a read directly from within the process. You have to make a system call to the OS and the OS is going to say this is the whole process that has made a system call and so, block the process till there is a packet from the network.

Student: if we do not need concurrency in this case then we can divide times of the running.

. So, clearly firstly there is really no concurrency in the system. Where is the concurrency? You can only run on one CPU, you block on a resource everybody gets logged on the resource. So, there two threads can physically never run concurrently, but

logically they can be made to lo like they run concurrently and that can be done by just for example maintaining a queue in some sense maintaining some kind of a scheduler inside the process.

And, the scheduler is going to say it is going to create multiple threads and it is going to say this thread get to run, that thread runs. And, then when it yields then you bring it back and we say now this thread you get to run and then when it comes when it says I want to yield, it yields and then you say another thread gets to run and so on. So, this is a way that a process inside itself is implementing a scheduler in sort of emulating multiple thread.

Student: Sir, what is the benefit of running (Refer Time: 46:24)?

Let us come to the benefit a little later, let us just understand how this is how can this be done using the abstractions; I want to talk about the benefits so. The other thing that that an OS does is a process should not be able to run away with the CPU. So, once a process has been scheduled on the CPU and OS should have a way of taking it back from the CPU. And, the way this is done is through what is called an interrupt.

So, if a process is running and an interrupt occurs the interrupt handler is the OS interrupt handler. And, the OS interrupt handler is going to say this process has been running for too long, let us just suspend it and get let some other process run on it . A similar method can be used at the user level using signals.

So, a user process can register a signal and it can say that the signal; so, there is another system call called alarm which allows process to tell the OS that a signal sig timer let us say should be generated every 100 milliseconds. So, every 100 milliseconds the sig timer hits a signal is going to get generated, the sig timer handler is going to get called inside the process.

The sig timer handler is maintaining this data structure of all the active threads in the system. It takes the running thread, suspends it, puts it in the ready queue, takes another thread, sets it running and returns. So, that way you also implement what is called preemption.

So, you can preempt a thread while its running and you can emulate a scheduler inside the thread. So, in some sense if you lo at the abstractions we have loed so far the signals abstraction of Unix is paralleling the interrupt abstraction that you see in hardware.

So, a hardware interrupt like a keyboard, key press or timer interrupt coming in or a network packet coming in these are all like activities that occur asynchronously and relatively rarely. And, you want to be able to and an OS is supposed to handle them and the way to handle them is what is called interrupts. So, an interrupt comes and the OS handle interrupt handler gets called, similarly inside a process a signal comes and the signal handler gets called.

Now, the question is what is the advantage of doing something like this? Well, clearly there is no concurrency achieved by having multiple user level threads inside. The advantage of kernel level threads, this is actually giving you physical concurrency.

The advantage of user level threads is that context switching is much cheaper for user level threads. If I want to context switch from one thread to another, let us say one thread says I want to give away the CPU, I want to yield. All it needs all that has happened is it will make a function call internal to the process and the process is going to take that, put it somewhere and set another one running.

So, there is no kernel crossing, there is no system call that is required which as you are going to see is relatively more expensive. So, there is you can do more things at the user level and you can do them faster. Further you can map multiple user level threads to multiple kernel level threads. So, you can the scheduler could be made intelligent enough such that you basically create multiple kernel level threads at the bottom. And, let us say you have two kernel level threads and you have five user level threads and its like saying that I have two processors and I have five processes and then I can map them anyway I like. And, I can have a completely custom scheduling policy depending on my application.

So, a kernel scheduling policy it has to be very general because it is going to run a variety of applications. A user level scheduler can have a very very custom scheduling policy which suits that vertical application. And, more importantly often software is written in a certain style and you want to be able to run it in different environments.

If your environment happens to be such that the kernel does not support, kernel level threads that is that used to be relatively common, still quite common. Then one way to be able to run the same program on in this environment is to be able to abstract it as a user level thread. The same code can run on this library where you are actually fooling the program to say believe that it has threads whereas, its actually running on a single thread, single kernel thread.

Student: And, during emulation also maybe emulating a different hardware also.

Sure. So, anything which involves running code that was written on a for assuming certain thing, but running in a different environment you can use this kind of a wrapper layer to be able to do these things .

So, we will stop here and continue next time.