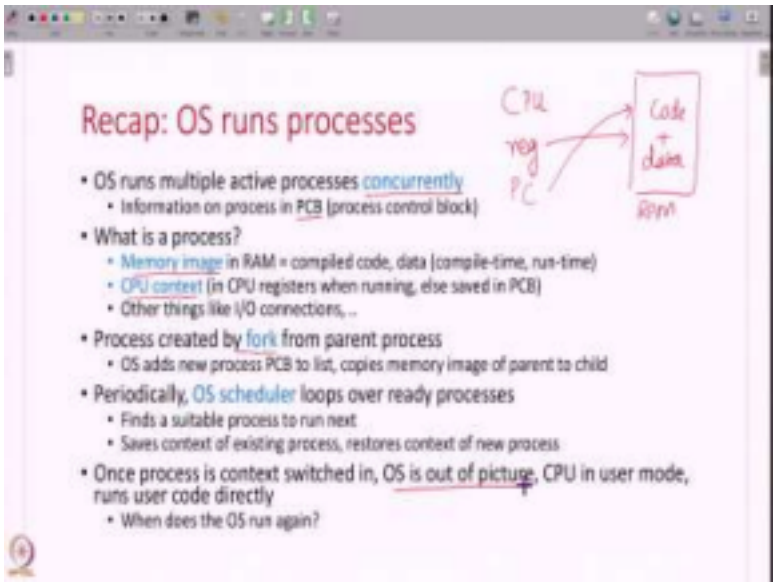
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture 07
Kernel mode execution

Hello everyone, welcome to the seventh lecture in the course Design and Engineering of Computer Systems. So, in the previous lecture, we have seen what are processes. In this lecture, we are going to go into a little bit more detail to understand how a process exactly runs. So, let us get started.

(Refer Slide Time: 00:34)



So, this is a recap of what we have seen in the previous lecture of how processes are run by the operating system. So, we have seen that the OS runs multiple active processes concurrently, it maintains all the information about a process in a data structure called the

PCB, and it periodically goes over the list of PCBs and finds processes to run.

And we have also seen that the process is defined by a few things, like its memory image in the RAM, there is the code and data of a process along with stack and heap and everything and this is called the memory image. Then, when a process is running, the CPU registers have information about the process, and that is called the process context.

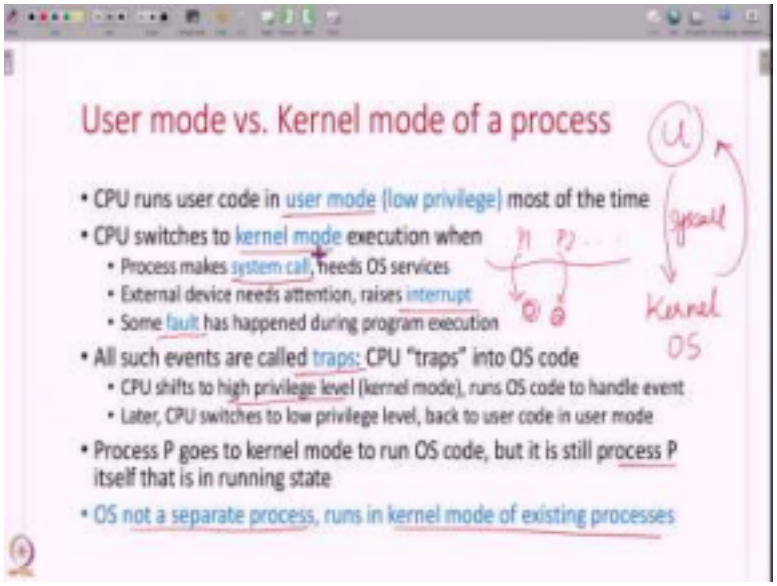
And when a process is not running, this context is saved in the PCB to be restored later. And a process also has a bunch of things like IO connections, which we have not discussed in much detail, which we will come to later in the course. And we have seen that processes are created by forking, when a parent does a fork system call a new child process is created.
And when the parent does fork, a new PCB is added to the list of processes.

And this child process also will get scheduled at some point by the scheduler. And the memory image is copied from parent to child and after that the child can do whatever it wants with its memory image. And we have seen the scheduler loops over all the ready processes from time to time to switch to different processes by saving the context of the old process and restoring the context of the new process.

So now, this is a high-level understanding of processes. In this lecture, we will go into a little bit more depth. So, for example, once a process is context switched in, the operating system starts a process to run its context is put in the CPU registers, the program counter is pointing to some code of the process, the registers have some data of the process and the process starts to run, it will execute one instruction after the other and so on.

A user program is now running on the CPU. And at this point, the operating system is out of the picture, it is involved in context switching a process starting it to run but once a process is running the OS is out of the picture and the user code runs directly on the CPU. So, now, when does the OS run again?

it has started one process and it has gone away behind the scenes. So, when does the operating system run once again, what causes the operating system to run once again?
(Refer Slide Time: 03:17)



So, that is the question we will try to understand in this lecture. So, when the CPU is running user code, we say that it is running in user mode, this user mode is a low privileged mode. And you cannot execute certain privileged instructions that access hardware and all of that. On the other hand, when the operating system is running, we are running in a privileged mode.

So, therefore, sometimes once in a while, the CPU will switch to kernel mode execution when it has to run operating system code. So, when does the CPU switch to kernel mode once a user code is running, when does the switch back into the operating system happen? It happens in mainly three scenarios.

One is when the process makes a system call, the user code says please read this data from disk or please write this to the monitor, display this on the monitor. Access to hardware is a privileged operation that the user program cannot do directly. Therefore, in such cases, a system call is made by the user process. So, when a user process makes a system call the CPU jumps into kernel code or runs the operating system code.

So, this is called kernel mode execution. When an interrupt is raised the user program does not need anything, but some external device has raised an interrupt and that interrupt again your user code does not know how to handle the interrupt. So, when the interrupt comes, once again, the CPU has to jump to some operating system code that knows how to talk to the external device and handle the interrupt.

Or the third case is, some fault has happened, your user program has made some mistake, access memory out of bounds something bad has happened during which the CPU says, oh no I have to go back to the operating system to figure out what to do. So, all such events

system calls interrupts, program faults all of these the generic term for them is traps, that is when any of these events happens, the CPU will stop running user code and trap into the operating system or their kernel code.

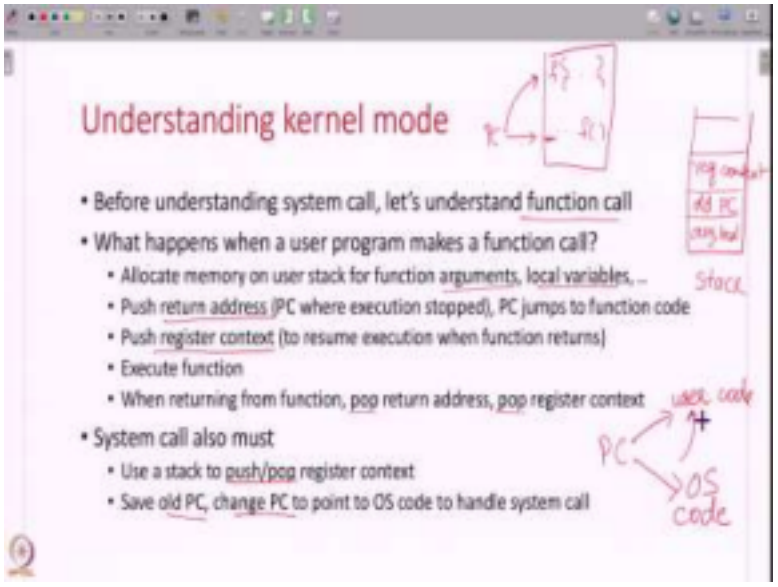
It traps into the OS code, that is why these are called traps. So, this basically involves shifting to a higher privilege level or kernel mode, running the operating system code to handle whatever event has happened. And after this is done, the CPU will not stay in the operating system code for long. The main purpose is to run user programs.

Therefore, once the event is handled, you will once again switch back to a low privilege level and go back into user mode again. So, periodically, the CPU which is most of the time running user programs will periodically jump into a high privilege level into the kernel mode, run operating system code and then go back.

So, when a process P goes into the kernel mode, you have a process P that is running in user mode, it goes into the kernel mode to run OS code. At this point, we say that it is the same process P that is still running, but this process P is running in kernel mode. So, an important subtle point to understand is that the operating system is not a separate process necessarily. But any process can go into kernel mode and run operating system code.

The operating system mostly runs in the kernel mode of existing processes. So, you have some processes P1, P2, and so on all of these are running on the CPU, periodically P1 will go into kernel mode to run some operating system code P2 will go into kernel mode to run some operating system code and so on. So, the operating system mainly runs in the kernel mode of existing processes, when one of these trap events happens.

(Refer Slide Time: 07:18)



So, now, let us understand a little bit more detail on what is this kernel mode execution? How does it happen? How is it different? We know a few things like privilege levels, but we will understand it in more detail. For example, what exactly happens during a system call? So, to understand this, let us first begin with a function call.

So, a function call is also moving to some other piece of code and running it, system call is also moving to operating system code and running it. So, they are somewhat similar, but there are also important differences. So, let us begin with understanding what is a function call? So, there is some code in your program, there is some function here and there is some other piece of code that calls the function.

And this is the actual implementation of the function. Initially, your program counter is pointing to the code here and this function is invoked by some code. At this point what happens your program counter will jump to the function code and execute those instructions and then come back. So, you all understand what is a function call or a procedure call. So, when this happens, what is happening on the user stack?

So, all of this is in the code, the instructions are there in the code section of your memory image. But what is happening on the stack? On the stack, things are getting pushed onto the stack all the time. And when a function call happens, things like arguments, local variables, all of these are also allocated memory on the stack. We have seen this before.

Then a few other things are also pushed onto the stack. For example, your old program counter or the return address here. This is also saved onto the stack. You push the return address onto the stack where you stopped execution. Why? Because now you are jumping, your program counter is being updated to a different location to run the function code. And

when the function finishes, you have to go back and resume.

How do you know where to resume? That is why you store the return address or the old value of the program counter on the stack. And you also store the register context on the stack. Some register context is also stored on the stack. Why is this? Because before you made this function call, maybe you loaded some values into general purpose registers. You are about to do some computation and then this function call happened.

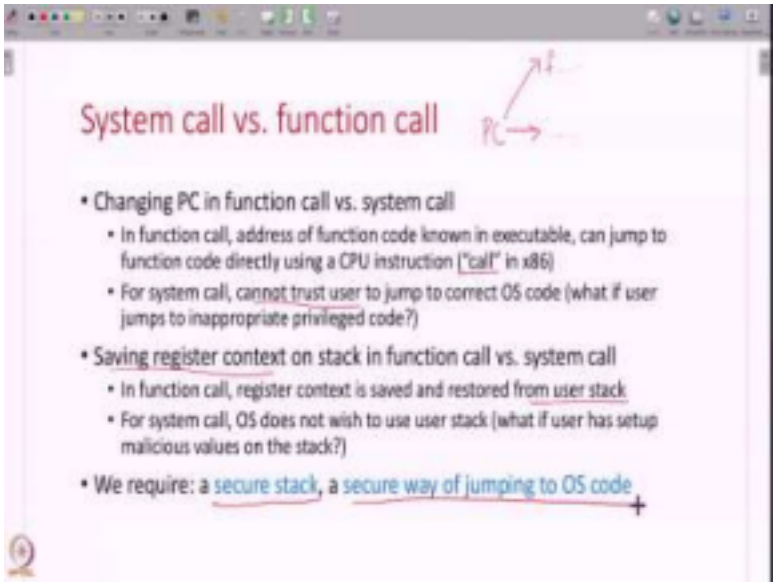
So, when you return back from the function you expect to continue whatever you were doing. Therefore, this register context also is saved on the user space stack when a function call is made. And of course, you will execute the function and after you return from the function once again, you will pop all of these things.

You will restore the register context; you will pop the program counter so that things return back to the state they were before the function call was made. Now you continue execution after the function returns. So, this is how a function call happens. In addition to things like arguments and local variables, you will also store things like the program counter and register context and everything on the stack, you will save it on the stack and restore it later.

Similarly, a system call also must do these things, it must use a stack in which you will push register context before the system call and pop it out afterwards. And you will also save the old value of the program counter and you will change the program counter to point to the OS code.

In a system call also at a high level, your program counter is pointing to some user code. And then you want to save this value of the program counter, you want to jump to the OS code that handles the system call. And then you want to go back again to the user code. This is also what you want to do in a system call.

(Refer Slide Time: 11:07)



But there are a few differences from a function call and what are those differences, let us understand them. In a function call, the address of the function code is known in the executable, and you can directly jump to the function code. So, there is a CPU instruction, for example, it is called the call instruction in x86.

So, what this call instruction does is your program counter is pointing to some instruction here and it will update the program counter to point to the function code, in an executable you know where the function code is located at which address you can easily jump to it, you can save the old value of program counter and set a new value for the program counter to start executing function code. So, this is easy to do, there is a CPU instruction for it also.

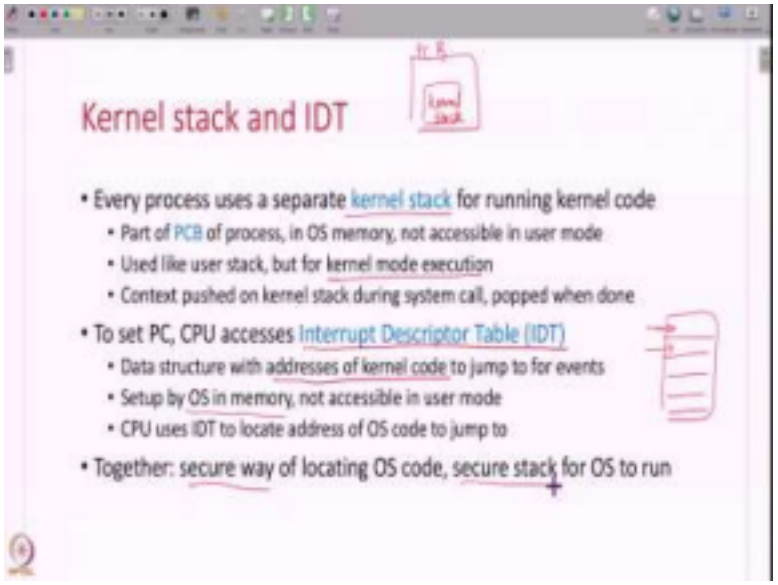
However, we cannot do the same mechanism for a system call. Why? Because there is a trust issue here, we cannot trust the user code to jump to the correct OS code like this for system call because OS code is privileged, it has extra powers. So, you cannot leave it to the user to say, hey invoke the function corresponding to a discreet, what if the user invokes a different function that is doing something malicious, we cannot trust the user.

So, therefore, the same mechanism of simply calling a function will not work to change the program counter for a system call. Then the next thing is saving this register context. When you made a function call you saved all of this register context on a user stack. But once again, when the operating system code is running, the operating system does not cross the user, it does not wish to use the user stack, what if the user has set up some wrong values on the stack.

The user can change his or her core to do bad things, to push and pop various other wrong things from the user stack. So, again, the OS does not wish to use the user stack to save and

restore anything when it is running operating system code. So, therefore, what we need is we need a secure stack and a secure way of jumping into operating system code. These are the main differences from a function call.

(Refer Slide Time: 13:19)



So, here are the answers, what do we do to solve this security issue when we make a system call, there are two important concepts called the kernel stack and the IDT that we will study now. So, every process in addition to a user stack, it also has a separate kernel stack. So, this is part of the PCB in the process control block of the process you have some memory called the kernel stack.

So, this kernel stack is like a user stack. It is used to store various things when function calls happen to store context and so on, but it is used only during kernel mode execution. Context is pushed on the kernel stack during a system call and it is popped when the system call is done, just like how you use a user stack during a function call you will use the kernel stack during a system call.

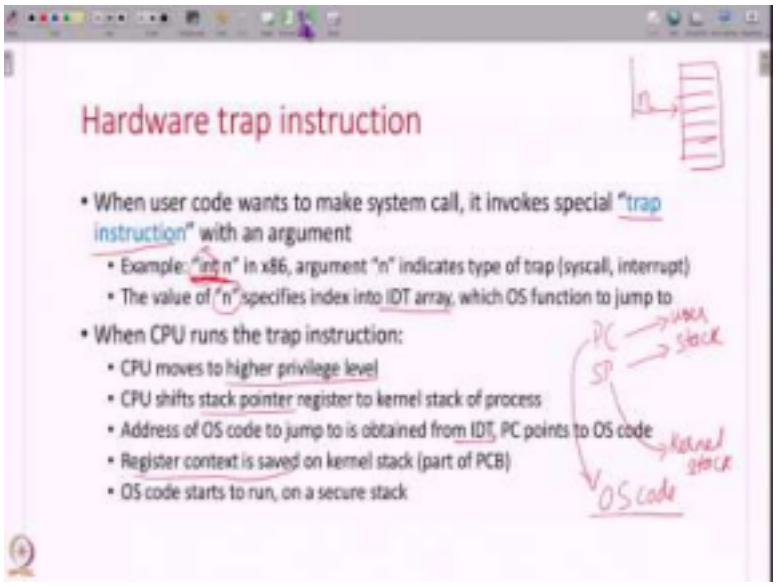
So, we will say that the context is saved on either the kernel stack or the PCB interchangeably because one is part of the other. And the other issue is how do you know which core to jump to, we cannot let the user decide which operating system code to invoke. For that, the CPU has a special data structure called the interrupt descriptor table.

When the operating system boots up, it sets up a data structure called the interrupt descriptor table. What does this have? This has the addresses of the kernel functions to invoke, if this event happens, you run this kernel code. Here is the address of a kernel code. So, this is an array which has entries for various events, if this event happens to go here, if this even

happens to go here, all of this is set up by the operating system in memory and the CPU knows this data structure where it is located.

So, whenever any event happens, a program fault, interrupt, system call anything, the CPU will look up this data structure and know which address to jump to what to set the program counter value to is known to the CPU securely. We are not depending on the user code or the user is executable to tell us where to jump instead this is supplied by the operating system itself in a secure fashion. So, we have a secure way of invoking OS code and a secure stack.

(Refer Slide Time: 15:41)



So, with these two things, we are now ready to understand how a system call or any trap works. So, when a user code wants to make a system call, it will call a special CPU instruction called the trap instruction. So, in x86, this trap instruction is called int n, int is the name of the instruction and n is an argument, this n indicates what is the type of the trap.

So, for any trap, system call, interrupt anything, anything that requires the operating system to come into the picture that is called a trap. And any trap begins with the CPU executing this special trap instruction with an argument. This argument is basically the index into the IDT array. So, the interrupt descriptor table has multiple addresses of, if this event happens run this code, if this event happens run this code.

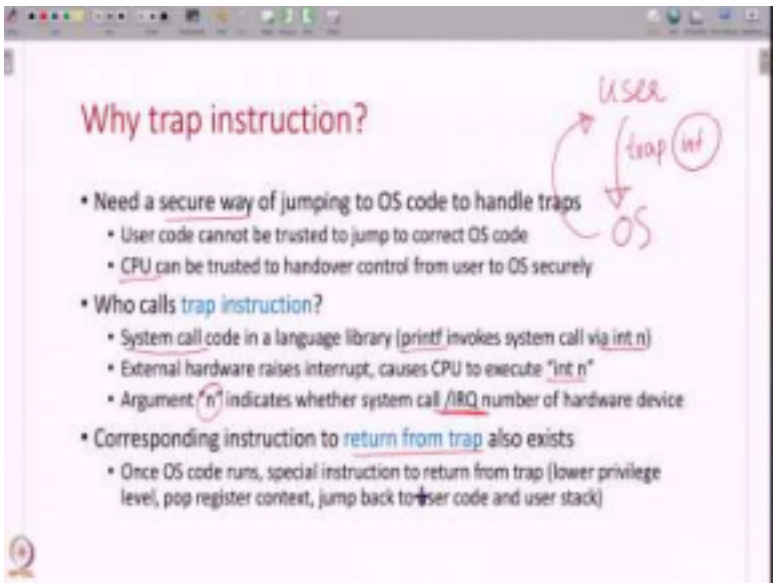
It has multiple entries, and this n will tell you which entry to use, and which operating system code to invoke. So, any event that has to trap into the operating system will begin with the CPU executing this int n instruction. And what happens when the CPU runs this trap instruction, it will move to a higher privilege level, it will shift the stack pointer from wherever it was pointing to before from the user stack to the kernel stack of a process and it

will find out what is the address of the OS code.

It has to jump to it will obtain that from the IDT and the program counter will jump to that, your program counter was pointing to say user code before, your stack pointer was pointing to the user stack before. When the CPU runs this int n instruction the program counter jumps to OS code, the stack pointer jumps to the kernel stack and at the end of this instruction and of course, on the kernel stack, once you are on the kernel stack, you will save the register context.

And at this point, all the previous context is saved. Now the operating system is ready to run. So, this is like a function call but a more secure way of calling into the operating system functions. So, after the trap instruction, now the operating system is in control. Until now, until the trap instruction has run before that only the user code was running. After the trap instruction, the scenario has changed; the operating system code is running on a secure stack.

(Refer Slide Time: 18:19)



So, you might ask why this trap instruction? Why do we need a separate hardware instruction? This is because as we have said before, we need a secure way of jumping into the OS code to handle all the traps and a secure stack. And both of these are set up by the trap instruction. And we cannot trust the user code to invoke the correct operating system code to run for any event.

And the only person we can trust, the only entity we can trust here is the CPU. The OS cannot trust the user to jump to the correct OS code. But the OS can trust the CPU to look up this interrupt descriptor table when an event happens and jump to the correct OS code.

We need control to go from user code to the OS code and this transition happens with this trap instruction or the int n instruction.

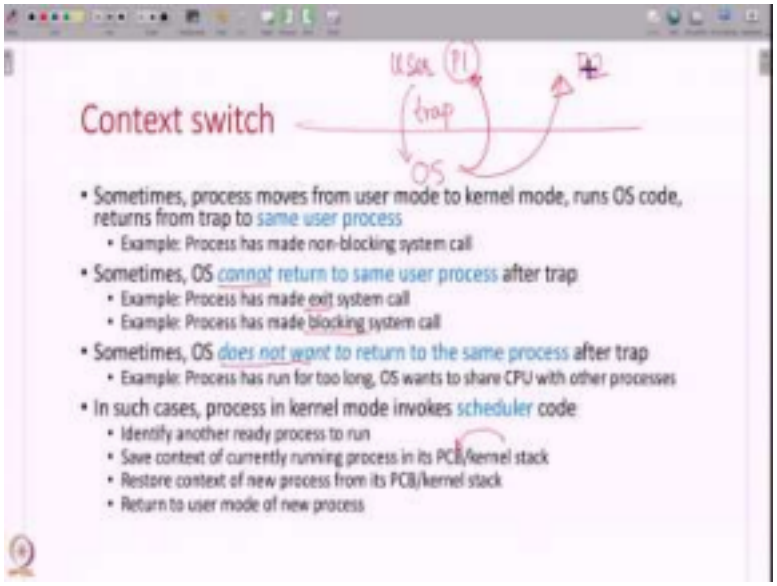
Now, who calls this trap instruction? Whenever you make a system call, you usually call into a language library. When you say printf, printf is actually a function implemented in the C library. And this language library will basically invoke this int n instruction. You as a user may not know it, but every time you are printing something to screen or reading a file from the disk or doing any of those privileged operations, you are actually invoking the int n instruction which is causing the CPU to tap into the OS.

This is usually hidden from your view by the language libraries. Then when interrupts happen, the external hardware will send a signal to the CPU and that will cause the CPU to execute the int n instruction. This is not triggered by the user, but this is triggered by the external hardware. So, whether it is triggered by the software or the external hardware, the end result is that whenever we want to jump into OS code, the CPU will execute this int n instruction.

And this argument n will indicate what type of an event is it whether it is a system call, if it is a different hardware device will provide different values of n. We have seen in a previous lecture that every piece of hardware gets an interrupt request number that is unique to that device. So, this n will indicate, oh this as either a system call or an interrupt from the keyboard or an interrupt from the disk and so on. And correspondingly, the CPU can jump to the correct operating system code.

So, this trap instruction jumps to the operating system code. Similarly, there is also another instruction called return from the trap, which is just the inverse of this trap instruction, it will jump into user code, it will restore context from the kernel stack, jump back into user code, switch the stack to the user stack, lower the privilege level, whatever was done by the trap instruction, it will undo all of that, there is also a return from the trap. So, when the OS is done running, it will call this return from trap instruction.

(Refer Slide Time: 21:24)



So, now, normally, what happens is in so far, the examples that we have seen a process is running in user mode, it is running some user code in user mode, then a trap has happened and it has jumped into the kernel code, it has jumped into the OS code. So, this is some user process P1 has trapped into the operating system.

Now, once this crap is handled, you can go back you can return from trap and go back to running the same process in user mode, that is possible. For example, if the process has made some non-blocking system call, requested some small piece of information OS will provide that information and go back.

This process has forked, OS creates a new child, PCB adds it to the list of processes and then goes back to the parent, that is fine. Sometimes you cannot go back to the running the same process after a trap. The OS cannot do that, why, what are the cases, what if this process has made an exit system call, once the exit system call is handled the OS cannot go back to the same process.

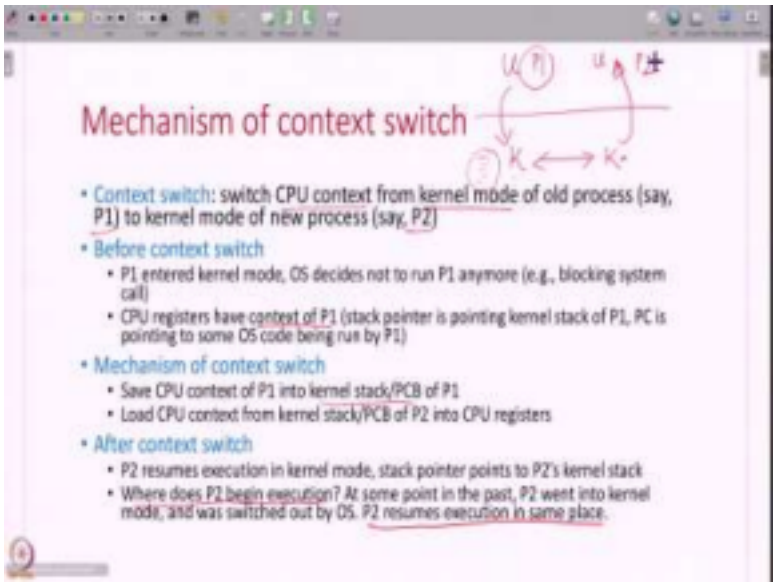
The process says that I do not want to run anymore or the process has made a blocking system call. It has said read something from disk in which case the process does not want to run till the disk data is ready. In such cases, OS cannot go back to the same process again. In these cases OS cannot return to this process.

And sometimes even if none of this happens, the OS may not want to go back to the same process even if the process is ready to run. For example, the OS scheduler after the trap, the OS scheduler thinks, oh this guy has run for too long, I do not want to run him anymore. In such cases also the OS may not go back to the same process.

So, sometimes you go back to the same process, say P1 made a trap you run over score you go back again in user mode to P1 and sometimes you might go back to a different process, say another process P2, that is called a context switch. In such cases, the scheduler is invoked and the scheduler will identify another process to run, it will save the context of this process P1 in its PCB or kernel stack.

The place where context is saved, the stack is the kernel stack and this kernel stack is part of the PCB. So, we will use these two terms interchangeably. So, if you do not want to run P1 anymore, then what do you do? You will save its context and then you will restore the context of P2. And this process is called a context switch from P1 to P2. So, when you are in kernel mode, sometimes you do not go back to the same process, but you jump to another process.

(Refer Slide Time: 24:10)



So, let us understand this mechanism of the context switch in a little bit more detail. So, the definition of a context switches. Your CPU context switches from one process say the old process P1 to the other process P2 and note that the switch is only happening in kernel mode, a user process cannot say switch me out, that is a privileged operation.

Switching processes is like underlying control of the CPU, that is a privileged operation, a user process cannot do it. So, a user process only when it goes into kernel mode, at sometimes when it goes into kernel mode, this process P1 has gone into kernel mode, then you will switch to the kernel mode of another process and then you will go back at some point to the user mode of a process P2.

This is how a context switch happens. So, before the context switch this process P1 has you

know had a trap or system call or something. For some reason, it has entered into the kernel mode and for some reason, the operating system has decided not to run P1 anymore. Maybe it exited it blocked whatever. At this point, the CPU scheduler is invoked. So, before the context switch, all the CPU registers have the context of P1, that is the stack pointer is pointing to some location on the kernel stack of P1.

The program counter is pointing to some operating system code that P1 was running in kernel mode. So, maybe P1 made a blocking system call to read from the disk and you ran some code saying give the command to the disk and so on. So, somewhere you are in the execution in kernel mode of process P1, this is the state before the context switch.

So, what happens during the context switch, you will save all of this context wherever you are stopping execution of P1 in kernel mode that context will be saved in the kernel stack or PCB of P1 and you will go to the PCB of P2 and you will restore its context, you will load the CPU context from the PCB of P2 into CPU registers.

So, now, what has happened after the context switch, P2 resumes execution in kernel mode. The program counter is pointing to some piece of code in the kernel mode of P2, the stack pointer is pointing to the kernel stack of P2 and so on. So, what is this context of P2 that is restored? Where does P2 begin execution? How do we know where to start the execution of this new process?

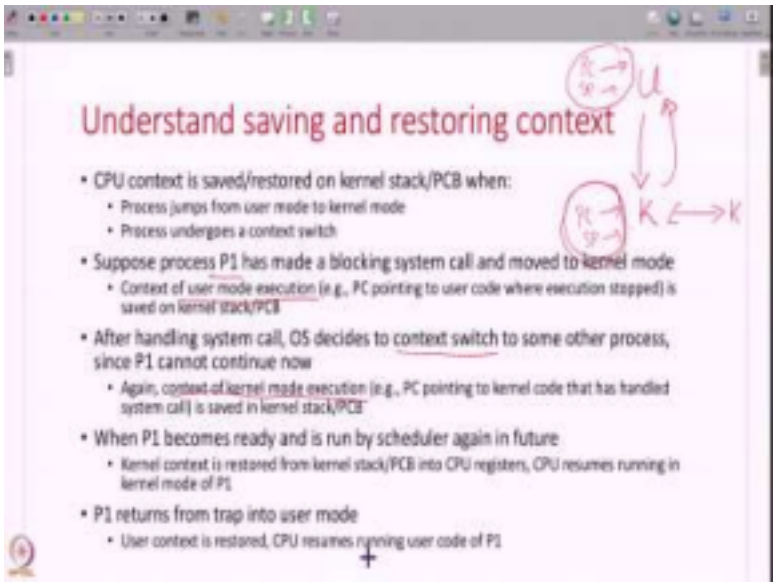
Recall that this process P2 also at some point went into kernel mode and was stopped by the scheduler. This P2 is not any different entity it is like P1, just like how you save the kernel context. The context of P1 how you have saved it in its kernel stack or PCB now, before stopping P1. Similarly, you would have saved, the OS would have saved the context of P2, P2 would have gone into kernel mode, would have been executing some code and it would have stopped in the past and that context is restored.

Therefore, wherever P2 stopped in the past at some point in the past, P2 went into kernel mode and it was stopped and it will simply resume execution in the same place. So, a context switch basically freezes this process and restores this process, resumes this process. Wherever P2 was stopped in the past, whatever context was stored the program counter, stack pointer, registers everything whatever was stored here that is restored and then P2 will resume execution in that kernel code where it stopped.

Maybe it made a read system call, maybe it gave some command to disk whatever and now the data is ready. P2 is ready to run and it will finish whatever it was doing and go back into

user mode again. So, this is a context switch.

(Refer Slide Time: 28:08)



So, just one subtlety around saving and restoring context. So, so far, we have seen multiple cases where this context is saved or restored. So, when a process jumps from user mode to kernel mode, we said that we saved context. We say remembered where the process was stopping in user mode. And similarly, when a process undergoes a context switch also we said that we save context.

So, understand that these are just two different contexts that we are saving. So, when a process P1 has made a blocking system call and it has moved into kernel mode, the context that you are saving here is the context of the user mode execution, your program counter is pointing to some user code, your stack pointer is pointing to the user stack, registers have some user data.

This was the state of the CPU and that context has been saved when you jumped into kernel mode. That was the context saved during a trap. But now, if you are doing a context switch after handling the system call if you decide to do a context switch then what is the context you are saving? This is the context of the kernel mode execution, where in the kernel mode you are pausing.

The program counter is pointing to some OS code that has handled the system call. The stack pointer is pointing to some location on the kernel stack. This is a different context and this context also you will save. And this is also once again saved in the PCB of the kernel stack. So, recall that context is not just one thing; every process does not have just one value of context.

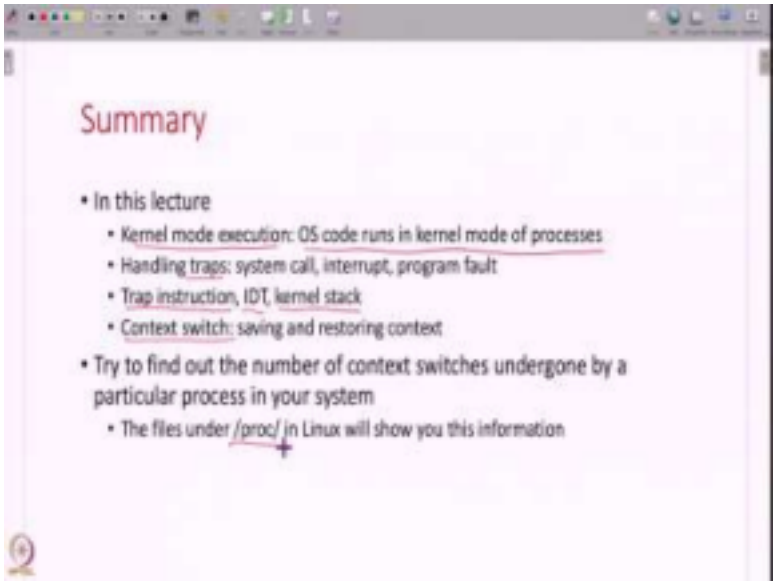
Wherever you stop a process, you stop a process in user mode; there is some context you have to save. When you stop the process in kernel mode, there is some context you have to save. This context also you will save and you will jump to some other process. Now at a later point of time the scheduler will come back to this process again.

When this process is ready to run in the future, maybe now it is blocking for disk, but once the disk data is ready, this process will be ready to run in the future. At that point, once again, you will restore this context; you will resume execution in kernel mode. And then after you are finished with your system call handling.

Now, your system call can complete. The data is available, you do not have to block anymore, you complete. And now you will once again restore this user context and you will jump back into user code. So, there is a separate kernel context, there is a separate user context. So, just understand that whenever you stop a process at any place, either in user code or kernel code or for a function call, wherever you stop, you will save context.

And once that event is done, you will restore context. So, therefore in your current stack or PCB, there will be multiple instances where you are saving and restoring context. So, this is a subtle point to understand when you have to really understand what we mean by saving and restoring context. So, with this, I would like to wrap up today's lecture.

(Refer Slide Time: 31:20)



So, in this lecture we have just gone into a lot more detail and understood what is kernel mode execution of a process. We have understood that sometimes or most of the times the CPU is running user programs and user mode. But once in while user processes jump into kernel mode and run OS code and this OS, the operating system, most of the time runs in the

kernel mode of existing processes.

And why does the OS come into the picture? It comes into the picture whenever there is a trap, either the user makes a system call or an external device raises an interrupt or some program fault happens. These are all traps. When these traps happen a separate a special trap instruction is run by the CPU, which will look up the IDT, jump to the suitable OS code, which will jump to a kernel stack which is a secure stack to be used when in kernel mode.

And once all of this is done, once the trap instruction and the trap handling is complete, you will go back to running user code again. And we have also seen what is the context switch. Sometimes when you come from the user code to the kernel code, you won't go back to the user mode of the same process, but you will switch to another process which is called the context switch.

So, these are some of the concepts in today's lecture. So, a small exercise for you is just look up your system to try to find out what are the context switches that are happening, how many context switches are happening at any point of time. If you are on a Linux machine, you look at the files under the slash proc directory. This is called the proc file system.
You can read up online on what this is. And some files within this directory will actually tell you how many context switches are happening in your system for each of the processes. So, this is a good exercise for you to get a feel for what we have started in toda's lecture. Thank you all and we will resume our discussion in the next lecture. Thanks.