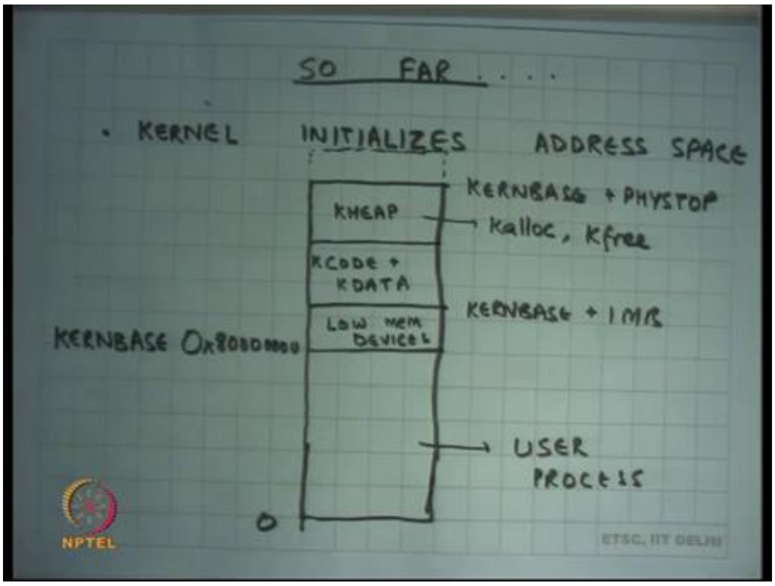
Operating Systems

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

Lecture – 17
Process structure, Context Switching

Welcome to Operating Systems lecture 17.

(Refer Slide Time: 00:31)



So, far we have looked at how the kernel initializes itself and initializes the address space and so, we have seen that the kernel basically initializes the address space such that it firstly, starts itself as at KERNBASE which on OX is this address hexadecimal 8 and 7 0. So, it is 2 GB right it is 32-bit address space. So, it is right in the middle of the 4 GB address space on 32 bits. The first 1 MB is for low memory devices it just maps it 1 2 1 2 1 2 the physical memory. In fact, that entire space here till from KERNBASE to KERNBASE plus PHYSTOP is mapped 1 to 1 to physical memory from 0 to PHYSTOP right.

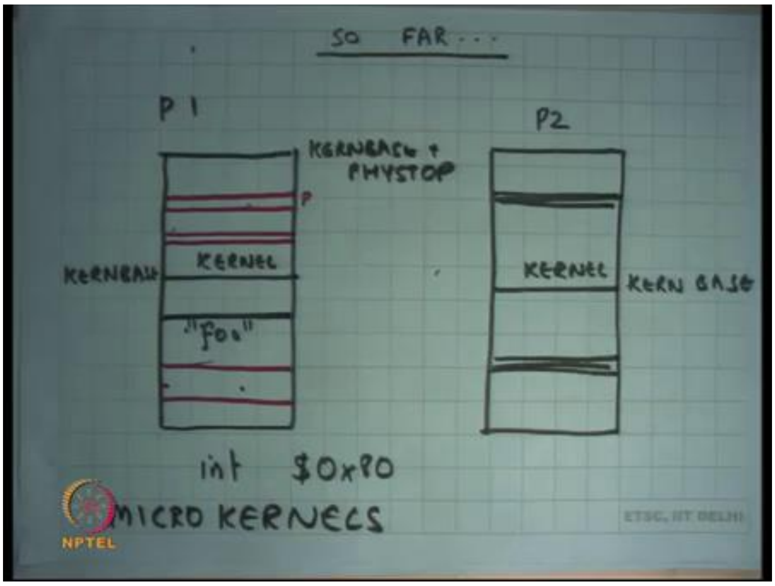
It loads the kernels code and data in this area right and the rest of the space which is available from this space onwards is what is called its heap and that heap can be is managed using these functions called Kalloc and Kfree right. So, if I want to allocate some space, I will call Kalloc and it will allocate a page from this space for me and that page I can. So, if you assume that will not be overwritten by anybody else and then Kfree is adding that back page back to the free less so that it can be used and so, memory can be reused and that way you can sort of not run out of memory.

And, this lower address space from 0 to KERNBASE is reserved for user processes so far we have not seen any user processes, but now we are going to talk about how user processes are created loaded and all that, we already have some idea the user process will be executing in this space. So, the user processes code and data will live in this space, the user processes stack will live in this space and user process heap will also live in this space right.

If a user ever makes a system call it will be executed like as a software interrupt instruction and the software interrupt instruction will dereference the IDT through the IDTR register and from that it will get the address of the handler. The kernel will set up the IDT in such a way that all the handlers are pointing in the kernels code and data region. So, the handler will execute in kernel mode right with kernels privileges right. So, the CS in the IDT will also say that execute this particular handler in privilege mode right.

So, this is how things are going to work from now on and forever really right as long as the system is going on alright.

(Refer Slide Time: 02:55)



If I have to sort of show this in more detail, let us say there were two different processes both of them will have the same kernel mapped. So, let us say this is kernel and kernel. So, above KERNBASE they have the same area mapped identically right from KERNBASE to KERNBASE plus PHYSTOP same thing. So, the entire physical memory is mapped of course, the process cannot access is directly because these are privilege pages and only code that is executing in privilege mode can ever touch these pages right.

So, the processes executing here it may have different processes may have different mapping. So, for example, in these red lines here in process P 1 let us say this page is mapped and this page is mapped and in this process P 2 let us say this page is mapped. So, they are different address spaces in that sense right and they are completely independent address spaces, also note that these pages which are you know which are present in the address space of the user or also coming from the physical memory right and the physical memory is also mapped in the kernel space.

So, the same pages in physical memory have two mappings in the virtual address space, one in the user side of things and another in the kernel side of things right. Also, where will be these pages be allocated from? From the kernels heap right; so, there is the kernels heap so, in the kernels space there is some space that you know static which is for kernels code and data and then above that all the other area is where I am going to allocate a page and create this mapping for the user process. So, that user process can implement its own address space right.

So, there will be all the user pages will have two mappings one which will be the users mapping. So, user can access it and one which will be the kernels mapping way for kernel to do it is own bookkeeping right. Of course, the user can only accesses page through it is user address, the user will not be able to access the page through the kernel address because the kernel addresses protected, is privileged right ok.

So, by ensuring that only the pages which belong to the user and ever mapped in this area the kernel basically ensure that the user cannot touch any privilege memory or the user cannot the process cannot touch another process memory alright. Also, we said that if once again if a process needs to make a system call then some code here is going to execute the interrupt instruction let us say it says int some number which indicates the system call number on Linux it is the hexadecimal 80 number.

So, interrupt hex hexadecimal 80 what is going to do is, it is going to go through the IDT and it is going to transfer control to an EIP somewhere here and that EIP should be the handler for that particular system call right or the system call in general. The user can give arguments to the system call by setting up its registered in a certain way, for example, the first register can say what is the name of the system call right.

So, you can say that let us say the fork system call is number 1, exec system call is number 2 and so on and that way you can basically set up set it up to say that this is the name of the system call and other registers can say give arguments right. So, integer arguments can be just given in registers for example, you know ECX can contain the first integer argument let us say a file descriptor**,** but if I was supposed to giving a string let us say the exec system call takes a string argument right and a string can be pretty large and registers are not enough to hold a string.

So, the way to do it is basically the string is going to live somewhere here right. So, let us say there is a file called foo and this string call foo is living here and so what the user does is he sets up the value of the pointer which points to foo as the first argument right in the register let us say ECX. And so, the kernel looks at this pointer and because the kernel has the same page directory map so, the same address spaces map when a de reference as a pointer it can read the string foo right.

So, this is important to understand when the user when the user makes a system call it is possible for the user to actually give pointers to the kernel to specify its arguments right. And, the reason the pointers work is because the kernel will execute the system call in the same address space as that of the program right. If the address spaces were different than pointers could not have been past.

So, this is the huge optimization I mean this is a big advantage of doing things in this way as opposed to doing things in a way you were using segmentation for example, because; segmentation will change the address space for the kernel or using a separate page table for the kernel right. So, you know often you may have wondered why I am eating up so much of space for the kernel in the virtual address space that way I am basically squeezing out the process right a process could ideally we had as big as a 4 gigabyte space on 2 to the power 32 bit machine.

But now you know these kernels like xv 6 or Linux or whatever else is basically constraining the process to live only in 2 GB right, but there is a big advantage to doing that. The advantages that the user and kernel can now talk by pointers, if I want to communicate some information I can just pass a pointer and that pointer works, if the kernel can also reply by a point by setting some value in the address spaces are user and give a reply. On the other hand, if the kernel had a separate address space then pointers would not have worked right ok.

Student: Sir.

Yes question?

Student: There is the physical space is supposing a 2 GB then can I allow a process to occupy a virtual space of greater than 2 GB right, if it was different also. If the stats were different, if suppose the physical memory is only of 64 MB. So, can I allow my process in my virtual memory to occupy space more than 64 MB?

So, the question is if my physical memory is small then can my virtual address space of a process be large, larger than the physical memory and the answer is?

Student: Yes.

Yes why?

Student: Virtual (Refer Time: 09:08).

So, you can make pointers first of all. So, you know the virtual memory does not need to be contiguous it can have you know noncontiguous mappings. So, even though the physical memory is contiguous the virtual address space is noncontiguous. So, some pages are mapped, some pages are not mapped, also you can map the same user process could very well say I want these two addresses to map to the same physical page right.

So, you can basically say you can have aliasing. So, you although it is not very useful to have you know 2 physical addresses 2 virtual addresses pointing to the same physical address in the user space itself, but would it be possible to do this. So, in any case so, I mean to answer your question it is possible for the virtual address space to be bigger than the physical address space.

Student: Then say in this case like you already know that the physical address space is possible by 2 GB. So, how is the kernel in my virtual space is distributed among my users?

So, the question is I mean if I assume that my physical memory is never going to be greater than 2 GB then this organization make sense right, but in real world that is not true number 1 alright. So, physical memory could be greater than 2 GB and still I am restricting my process to have only 2 GB at most if you are using xv 6. If you are using Linux, you are restricting to 3 GB right that is not a great idea great design I mean today we have memories of 64 GB even 100 and 28 GB.

So, restricting that a process can only access 3 GB basically means if I want to actually write a program that needs that much memory, I need to have a set of processes. If I am executing on 32-bit machine right, on the other hand if I am executing on a 64-bit machine this problem gets solved automatically right ok.

Student: Sir.

Yes.

Student: Sir it says anyways there for every address in the user mapping there are two corresponding virtual address which map to the same physical address that is one in the kernel and one in the user sir. So, as like so if there was not a separate. So, if kernel was not living in the user address space if kernel was living in the separate space, but, but suppose and it had to, and the user have to make a system call. So, could not they be a mechanism in which the user could have passed a pointer which contains the kernel mapping instead of it is own mapping.

Ok. So, the question is could not the user. So, I said that the user can plus pointers to the kernel, it can set up some area in it is own memory, it can initialize some values in it is own memory and use the address of that memory location and pass it to the kernel and the kernel can dereference it and because the kernel is executing the same address space it is to dereference. And the question is if the kernel was executing in a different address space could the user have figured out what the pointer is in the kernels address space to the location that it has set up and pass that value to the kernel that is very difficult right, how will the user know, what is the value of the kernels pointer in that address space right.

So, you know you need some interface to basically it will be able to communicate that and if you have to do it for all the addresses in the user space that is very costly also right. So, you know these are interesting ideas questions and you know something to think about on your own for some time and then we can discuss about it more right. So, lot of thought has gone into you know what the right operating system design is and some most.

So, the mainstream kernel that we use today have taken this design because of it is efficiency right because the communication between user and kernel is very fast just exchange a pointer right and that makes things and you know speed is by far the you know has by far been the most crucial factor in deciding a design. They have been other operating systems which are actually popular not a necessarily in the mainstream desktops and you know space and server space. But, maybe in the embedded devices space where security is more important for example, right and those kinds of organizations what is called Micro kernels right.

Here the idea is that the kernel subsystems live in separate address spaces and so, this gives them very strong isolation from each other right. So, for example, there will be one process which is the system call handler right and so, if you want to make a system call what you do is you just pass the arguments. But, now this you need to copy your arguments from one address space to another address space which and the other process which is doing the system call handling is going to read those arguments and then execute it right.

So, these are all organizations, but I mean they have their pros and cons if you basically divide the kernel into separate address spaces it gives you very strong protection, isolation, but it decrease a performance right. So, that is a tradeoff question.

Student: Like in case P 1 user users kalloc the kernel allocates makes a page and allocates to foo right. So, sir that process that address physical address will be mapped in user space and it will also have a mapping in kernel.

Student: So, but how will the kernel of P 2 will know that, where is that mapping, because we are not actually updating the page directory of P 2 right.

Why does so, why does the kernel of I mean. Firstly, there is only one kernel right it does not mean there is no kernel of P 1 and kernel of P 2 it is one kernel which is mapped common.

Student: P directory.

Or yeah let us say if you are access if you are working in the address space of P 2, then why does not?

Student: How does?

Why does not need to know?

Student: Sir, because next time suppose P 2 ask to allocate for another page, how will it know what space does we use?

Alright. So, these mappings in the kernel side of things remain even in P 2 right these mappings get removed, but these mappings still remain, and these allocation data structures also remain. So, it knows that these areas have been allocated already.

Student: Sir but suppose when the P 1 was running and it ask for kernel to allocate a page, it allocated page and updated it is entry the free another details free pager.

Right and so, the date structure the free less itself living in the kernel space right ok.

Student: Sir, but it is it understood.

Right. So, free less itself is in the kernel space and so, even if you switch to P 2 the free less remains just where it was right. So, all the data structures to do all this book keeping are living in the kernel space and because kernel spaces shared between all the processes you basically have you know you do not information is not lost the kernels information remains, question.

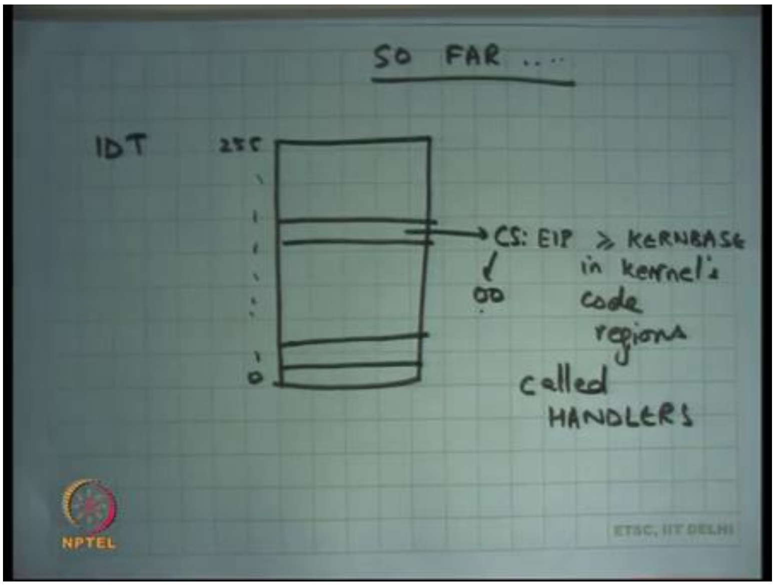
Student: Sir, user page in the allocated from kernel heap. So, why in the system like Linux why do we need 3 GB for user address space and only 1 GB for kernel address space?

Ok, very interesting question, if the pages in the user space are allocated from the kernels heap then should not the kernels heap be necessarily larger than the size of the user space really. So, does not need to be larger because as we said that you know the address space can be bigger the actual allocation size can be smaller. So, you only say you know let us say you the address space of the process could be 3 GB, but the allocation size could be let us say only 5 MB.

If you wanted to make larger allocation sizes let us say greater than you know 1 GB or 2 GB then as we said before you know xv6 is a very simple operating system which requires that the entire physical memory is mapped in the kernel address space, but modern full operating systems like Linux or you know anything else is basically going to not going to map the entire physical memory in it is address space it is going to recycle the address space and make it point to different regions and it is going to do book keeping and other physical page level that which pages have been used and which have not been used right.

So, it is not necessary xv6 is one an example often operating system which maps the which requires that the entire physical spaces mapped in the kernel address space. But that is not necessary right, it makes thing simple and that is why we have discussing it first.

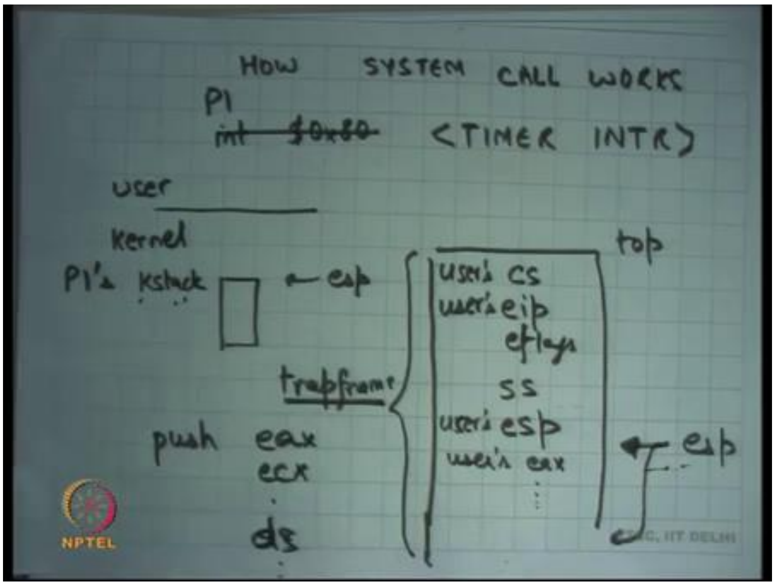
(Refer Slide Time: 17:23)



So, we have already seen this there is an interrupt descriptor table that has CS colony IB pointers CS has the last 2 bits as 0 0 which means it runs in privilege mode and EIP is somewhere in the kernels code regions so the handler gets called right. And, these pointers which are so initialized these are initialized by the operating system itself and we have already seen that the user is not allowed to overwrite either the IDTR which points to the IDT or the entries in the IDT itself.

So, the IDT itself is going to live in the kernel space so, clearly the user cannot access it, if it was able to access it then it can just bring down the system.

(Refer Slide Time: 18:03)



So, let us look at how a system call works alright. So, let us say this is user space and this is kernel space, and this is the process P 1 right and in the kernel space every process has a stack. Now let us assume xv6 is processes model so, every process has a separate stack kernel stack P 1’s Kstack right.

When the process makes uses the interrupt instruction to make a system call immediately the stack shift it is the stack points to stacks pointing to Kstack right and the hardware pushes some things on the Kstack. What are those? Let us say cs eip these are users cs right users eip whatever were the old cs, whatever were old eip, e flags. So, whatever were these value of these registers before the interrupt ss and esp right and so, these get staved on the. So, let us say this was top of the kstack.

So, now the new esp points here right. So, let us say this is users esp right. So, that is what happens immediately after you make a system call as soon as interrupt instruction is executed the state of the architecture starts looking like this cs, eip, eflags and stack has shifted. So, whatever were the users stack that is of no consequence it has been just mean saved whatever that is the that is represented by esp and this esp must necessarily will be a pointer in the users address space. Similarly, this eip must necessarily be a pointer in the users address space right ok.

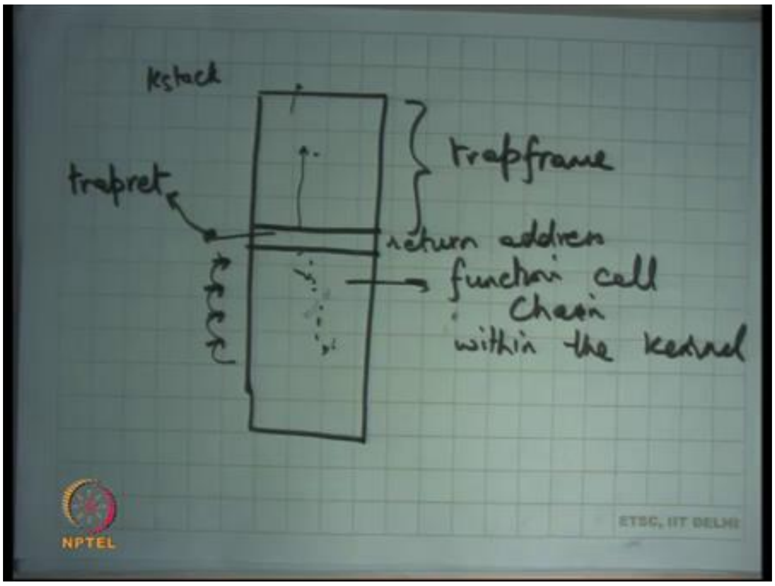
So, now the handler gets called so, at this point the handler gets called and the first few thing it is going to do is it is going to say you know execute instructions like push other register let us say push eax ecx, you know other segment registers let us say ds and so on right. So, it is going to push all these registers and so, eventually at some point esp is going to come here and you going to have let us say users eax and so on, on the stack right.

So, the kernel stack is a nice place where we can save all this information about what the values of the user registers at the time of this interrupt instructions were right. And so, this entire structure is call the trap frame it contains all the registers that have been saved by the handler and it contains all the registers that was saved by the hardware right and this trapframe is important.

So, the reasons called the trapframe is because it is the frame that is initialized on every trap right and the reason it is important is because from here and the kernel functions can figure out what were the values that the user registers at the time of the interrupt system call. So, example if you if you wants to know the arguments it just needs to looking to the trapframe also the return value can be pass to the trapframe, if I want to pass the return value in the eax register then all I need to do is just overwrite the trapframe eax right.

At the time of return from interrupt this trapframe will get popped in the same order in which it was pushed in the in the opposite order in which it was pushed and so, first few values will get push popped by the software and then the I rate instruction is going to pop the last 5 values. And the so, if the user will continue as though it was it never went into the kernel, except that the return value would have changed and may be something some other things would have changed depending on the semantically system call alright.

(Refer Slide Time: 22:13)



So, basically if I want to draw the kstack at any point it will initially have the trapframe right, then it will have some other you know whatever function call chain within the kernel right. So, the handler will push all these registers and then it may call some function right depending on what vector it was for example, it calls cis call function. The cis call function will read the first argument and let us say it was the it was the exec system call. So, it will call the exec system exec function or something right. So, all these functions their function call frames are also saved on this stack right.

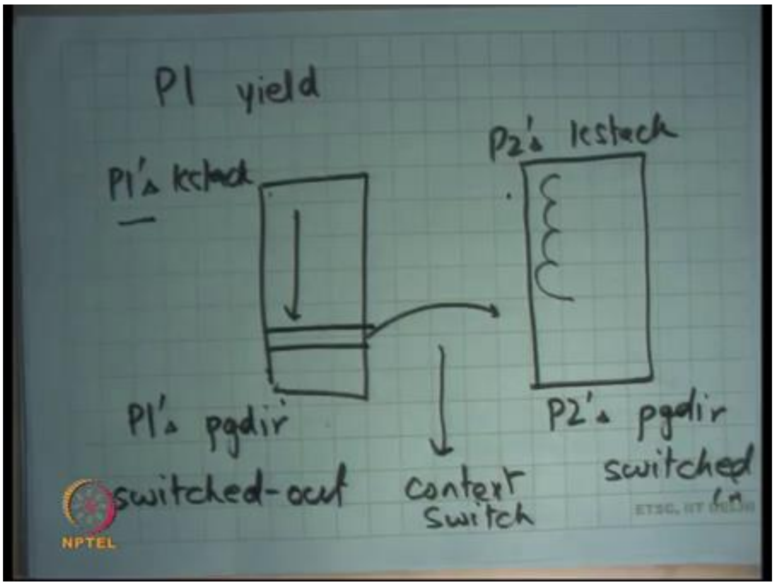
So, you basically keep pushing this stack like this alright and at some point you would have a handled the system call which may involve overwriting values of the trapframe and then you will start returning from the function just like you return from normal function calls right and at some point you will return from the trapframe return from the kernel to the user. And so, the last function that should be kept called is a function that pops of the trapframe and then calls iret so, to return to the user mode right.

So, on xv6 this function is called the trapret function right. So, in other words what should happen is that whatever is the function call that you called the last return address should be trapret. So, when the last function returns it should return to trapret and trapret will basically just pop of all these registers and then call iret to return back to user mode right. So, this stack does not only does not only contain the values of the registers it also contains the values of the instruction pointers that should get called and in what order right, will stack basically acts as a function as a function stack and so, any every time you return you basically pop of in a value from the stack and jump to it right.

So, if your stack contains a reference to trapret then when you return from the function chain you are going to next execute trapret and trapret is going to do all this for you and then you are going to jump back right. So, in other words the kstack the kernel stack of a process contains enough information to say watch, where should you start a executing and what all should you execute next and with what values and eventually it should typically just go back to the user after it has finished the execution alright.

Yes, question. The question is, will trapret pop the entire the trapret frame and then call iret? No, trapret with pop only the registers that was saved by software right and the last 5 registers will be pop by iret alright. So, this is so, the important thing is that the entire information about the kernel side execution of the process is encapsulate in the in the stack kstack right. So, let us say a function wanted to call the yield system call right.

(Refer Slide Time: 25:49)



So, far we have seen a system call that that winds the kstack and then unwinds the kstack and then returns to the user mode, but yield system call which requires you to switch the process right, it is basically saying I do not want to run any more let somebody else run right. So, yield an example of a system call which will enter the kernel on P 1 kstack, but exit the kernel on P 2's kstack alright and in the middle this switching of stacks is what is called a context switch right. So, basically when a process calls yield that is a P 1 calls yield P 1’s kstack will get formed the entire chain will get saved on the kstack.

And at some point it is going to say let us switch the stack alright and so, you going to go to P 2’s kstack you going to go to P 2’s kstack and instead of unwinding P 1’s kstack as in the previous example I am going to unwind P 2’s kstack and P after and when I unwind P 2’s kstack the re and also I am going to change to P 2’s page directory right.

So, this was P 1’s page directory. So, what happens? So, this process of shifting from one processes kstack and one processes page directory to another process kstack, and page directory is called context switch. So, you change the kstack from P 1 to P 2 you change the page directly from P 1 to P 2 and then you just execute on this kstack and then when this kstack it is unbound just like before. So, by returns you actually end up in P 2 exactly as it was executing the last time it got preempted right.

So, what has happened is, this kstack will get saved in memory right. So, this kstack will get saved and this kstack will become active at some later point if let say this kstack process called yield and then it wants to shift to this kstack all it needs to do is change to this kstack change to this page dir and now this will get unbound just like a what got the unbound even if there was no context switch right.

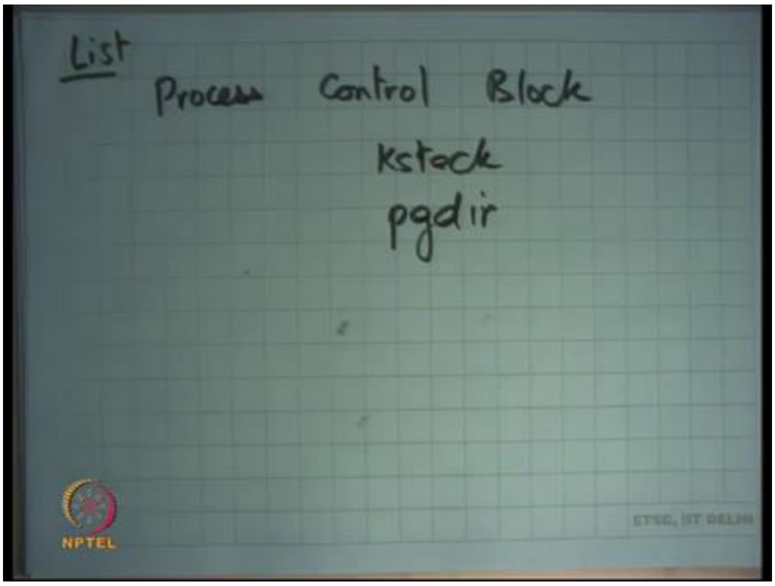
So, that is how a context switch works one process make a system call you execute in the kernel you form some state in the kernel which is saved in the kstack, you switch the kstack, you switch the page directly, you unwind the kstack of the other process. And your back you are suddenly executing exactly as you left the other process in some previous iteration of these procedure right.

So, the page directory of a process and the kstack of a process contain the state of the process the running state of the process as it was left the last time it was context switched out right. So, this process is called switched out process right and this one is called switched in right. So, the kernel maintains a list of all the switched out processes, the scheduler of the switched out processes and switched out process must have a kstack and a page directory, you switch to the kstack and you switch to that page directory and you are now running in that particular processes mode. So, that is how context switch happens at the process level.

Student: The switch that list is PCB s.

Yeah so, these list of you know this list which contains all these process states is basically the list of PCBs right.

(Refer Slide Time: 29:51)



List

Process Control Block

kstack

pgdir

NPTEL
ETSC, IIT DELHI

So, typically the process control block will have pointers to it is kstack and pointer to it is page directory and so there is a list of PCBs. So, this is a list of PCBs and the scheduler will pick one PCB and then switch to it is kstack and it is page directory and that is it, that is a context switch. So, all the information that was necessary to encapsulate the information exactly where that process got context switched out is available in the kstack.

Student: Why cannot save the pointer to the page directory on the stack?

Why cannot we save the pointer to the page directory on the stack? Ok that is a very interesting question, can you, can you not, well actually in theory you can nothing stops you, you just out to make sure that it never gets overwritten right. Recall that recall that when there is a there is an interrupt immediately you switch to kstack and things get push on the kstack.

So, you would make sure that the top of the kstack that you initialize that you put inside the TSS the trusted segment is not such that you overwrite that page directory. So, you can use one you know that top plus one area to basically contain the page directory also right. Is it a big win? Perhaps not, because I mean what are you saved you know you still using the same amount of space overall you could have saved it in the PCB you have saving it in the stack perhaps complicating things a little bit yourself right.

Student: Sir, I think sir since a different process share the same kernel space. So, the case same kernel mappings so, the kernel stack is also same in both.

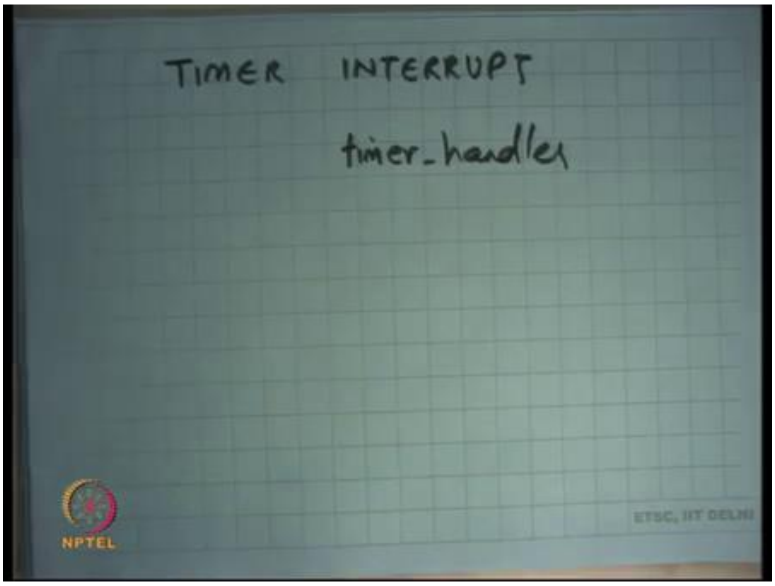
Ok. So, here is an interesting thing, because each process has the same mapping of the kernel everywhere when you context switch the page dir it is right. So, the nice thing about this picture here is that in this context switch when you shifted from P1 page dir to P2 page dir, because you are executing in kernel mode. And the kernel is mapped identically in all the address spaces there is no problem, you know it is not like the eip has changed or the esp has changed or the context at eip have changed or the context at esp have changed all the values currently are kernel values.

And, even if you change the page directories you are right you could not have done this for example, if you are executing in user side right because you have different, different address spaces the movement you change the page dir you know your eip will no longer be valid the next eip right. But that does not mean that they have the same page dir right I mean it is a same space from which they have allocated different regions for different processes.

So, one case so, in the same Kheap there will be one page for P 1’s kstack, and another page for P 2’s kstack, there will be yet another set of pages for P 1’s page dir right and yet an another set of pages for P2’s page dir, they have to be completely disjoint structures right. It is in the same space, but they have different areas that have been allocated for different processes good ok.

So, we have seen we seen how yield works right and yield is an example of a process saying I want to give up the CPU I want to be a good citizen I want to give the CPU to somebody else. But, we have also said that you know if a process is not yielding the CPU and the process does not obliged to yield the CPU really then the OS should have the mechanism of taking away the CPU from the process and the way to do that is the timer interrupt right.

(Refer Slide Time: 33:45)



TIMER INTERRUPT

timer_handler

So, we said that the OS can you know there is a device called the timer in the hardware and the OS can configured it is frequency can say the fire every xv number of milliseconds and so each time the timer interrupt fires it has a same effect of an interrupt right. So, once again the IDT gets consulted and the appropriate handler gets called, earlier it was the cisca handler that was getting called this time it will be the timer hander which will get called let say right once again the mechanism will be identical on.

So, instead of interrupt it 0 x 80 let us just say an external timer interrupt occurred right and once again P 1 will shift to kstack and it will push all these values on the. So, these 5 values will be pushed by the hardware, other things will be pushed by the timer handler and now it will call the scheduler alright.

And, will be the identical thing I mean just like yield I mean the only difference between a timer interrupt and the yield system call is that a timer actually forced the preemption, yield was a voluntary preemption the mechanism is identical alright. Once again the kstack will contain all the information about exactly where the process got preempted and what where the values of it is resistors in the trap frame right and the kernel will execute whatever the kernels call change that will also get saved in the trap frame.

And, then you context switch just like before right you will context switch the another processes kstack would be saved, the other process may have been preempted by a timer interrupt or may have been preempted because of yield call in either case the unwinding process is similar and you go back the user right, that is how and an involuntary context switch happens. The previous case was a voluntary context switch we have somebody called yield this is an involuntary context switch where the time interrupt occurs and the scheduler decides that you need to be switched off or you have been running for long.

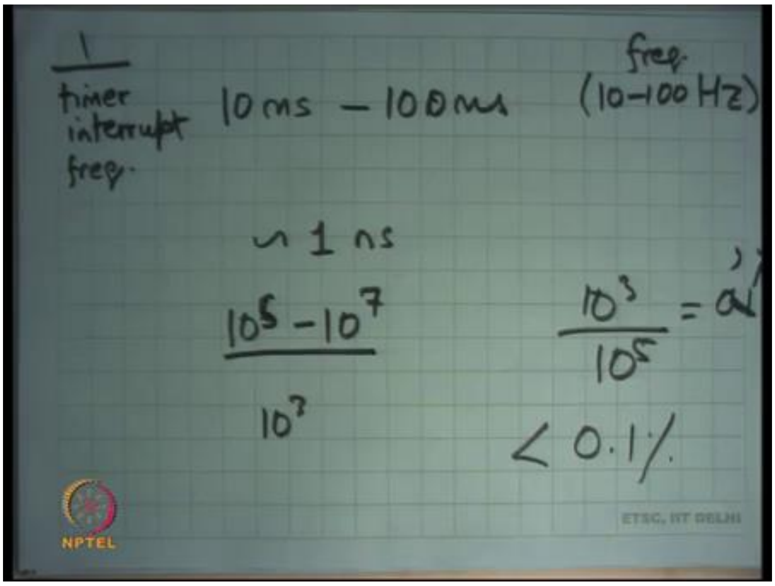
Student: So, the handlers are same for both.

Are the handlers same? No, I mean they have roughly calling the same functions at the end, but initially they are slightly different ok, one is the system call, one is not a system call alright very good. So, now, we understand how timer interrupt works, let say let say the time interrupt occurred and the scheduler decides that it does not want to context switch. That is a possibility, no problem you will not switch the stack, you will just unwind the same stack again and you are back where you where you where you interrupted alright.

So, one question that you may have is this you know this whole business of actually a timer interrupt coming and you know this kernel code getting called is an unnecessary nuisance. If there is only one process that is going to run you know let us say your running some high performance computation we have written some very fancy program and you are running it and you know it is a only program that is running in the system.

And, now this timer interrupt comes and bothers you every time and only for the scheduler to say oh this is the only process running and I am just you know I just wind the stack and then unwind the stack and what is the overhead of this right. So, the overhead depends on what, on the frequency of the timer interrupt right, how fast is the timer interrupt occurring right.

(Refer Slide Time: 37:33)


1
timer interrupt freq.
10 ms – 100 ms
freq. (10-100 Hz)
~ 1 ns
10^5 - 10^7
10^7
10^3 / 10^5 =
< 0.1%
NPTEL
ETSC, IIT DELHI

So, let me just first tell you know what kind of members are there, let us say 10 milliseconds to 100 milliseconds is the duration between 2 timer interrupts right. So, or 10 to 100 Hertz is basically the frequency typical frequency that you use for your timer interrupt right. On modern machines one instruction so, this is timer interrupt this is for one by timer interrupt frequency alright.

So, say 10 to 100 Hertz alright and this frequency and how long does it take to execute 1 instruction on a modern processor let say you know 2 point some giga Hertz or whatever. So, it is you know roughly on the order of 1 nanosecond right. So, how many instructions can execute in one timer slot? Let us say 10 to the power 6 to 10 to the power 7 instructions right on let say 10 to the power 5 to 10 to the power 7 even if you want to be you know if you want to say that memory accesses going to take lot of time, you know memory is not so fast or whatever.

So, 10 to the power 5 to 10 power 7 right. So, that is the number of instructions that get executed roughly speaking in one timer slot, what is the overhead of one timer interrupt if you would not be going to context switch. It is you know one switch into the kernel executing a few function calls returning from a few kern function calls and going back, how many instructions do you think will get executed in doing this.

Student: (Refer Time: 39:16) 100.

So, somebody says 100.

Student: 20.

20 let us say you know 1000 is you know a very conservative number. So, 1000 instructions so, that is basically let say 10 to the power 3 right. So, the overhead is something like 10 to the power 3 upon you know very conservative figure 10 to the power 3 upon 10 to the power 5 that is 1 percent right, actually it is going to less than 0.1 percent you know actually you know.

So, this is 1 percent of course, but roughly the overhead is less than 0.1 percent of doing this timer interrupt on modern hardware alright. Of course, you know you may say oh, but so, what happened in the early days you know. So, we right now 2 giga Hertz machines. Let us say not very too long back let us say 20 years back we had only 100 megahertz machines maximum right. So, when you had 100 mega Hertz machines this overhead may have increased because the number of instructions you can execute is smaller you know time interrupt.

But then you know you also used you know larger values of the frequency let us say 100 milliseconds. In fact, you know it you can safely decrease the timer interrupt frequency from 10 milliseconds to even 1 millisecond and still not see any overhead right, but you know there is no real advantage of doing that and exactly why you will why you know what other factors going to designing what the time interrupt frequency is or we going to discuss as we go along the course.

Student: Cannot we change the frequency based on the number of processes running we already have the information how many processes are running. So, can reduce or increase the frequency accordingly.

Should not the timer interrupt frequency depend on the number of processors that are running mostly you by know.

Student: Thank you so because it is up to scheduler to.

Firstly, let us understand that this timer interrupt is being generated per processor right. So, every processer has a separate timer device attached to it in theory right, how it works in physical hardware is the different thing, but let us just assume that each CPU has a separate timer device attached to it, each CPU is doing it is own scheduling basically right and of course, the scheduler is a common scheduler that understands that there are many processors in the system.

And, it basically chooses whether you know this process is supposed to run on this CPU or not etcetera right. So, every CPU has one timer device so, I mean the timer interrupt frequency should really have no relation to in the number of CPU’s in the system right each. In fact, each CPU can have a different timer frequency for that matter, does not make sense may be not, but you know it is possible.

Student: Sir, would not it become hard to would not it become hard to than synchronize all of these things.

Would not it become hard to synchronize all these things if each of them had a different timer inter frequencies, what do you mean by synchronize?

Student: Like one of them working at 10 milliseconds other is at 9 milliseconds. So, when I.

So, the question is do these processors needs to be synchronized, is it necessary that each processor gets the timer interrupt exactly the same time and comes back exactly at the same time, no not at all alright. So, this processor could be executing you know could get a timer interrupt now and another processor could get an interrupt 9 milliseconds from now perfectly ok, does not matter ok, they do not need to be synchronized with each other.

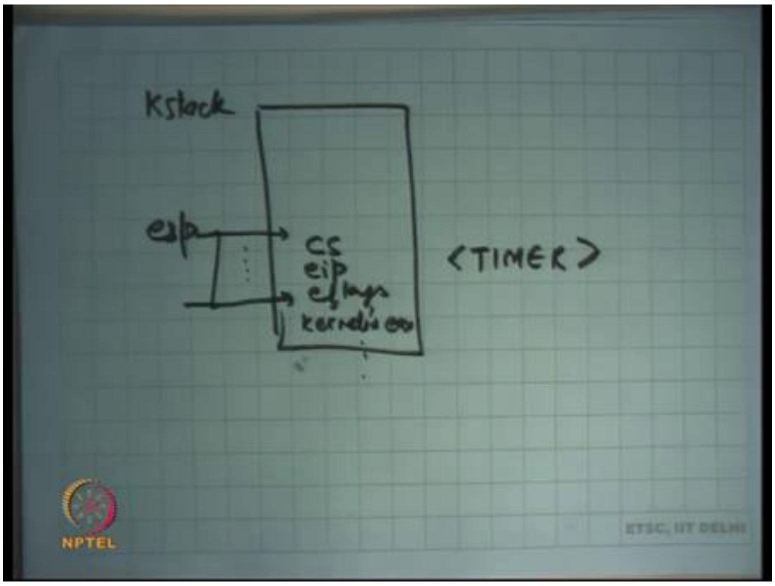
In fact, synchronization will be very hard to implement in hardware you know that level of synchronization is almost impossible to achieve ok. So, they do not need to be synchronized the timer interrupts are separate come I they act as completely asynchronous devices. Finally, I gave you an example where the timer interrupt occurred while the process was executing in the user mode right. So, I gave you an example where I said, let us see right.

So, I gave you an example that P 1 was running and the timer interrupt occurred in the user mode right and so, this all this started happening after that right, but can a timer interrupt occur while the process was all processor was already executing in the kernel mode.

Student: Yes

Yes why not, I mean the timer interrupter has no idea whether you are executing in kernel mode or not, it is possible for the kernel to disable interrupts while it is executing the kernel mode, but on xv 6 it does not do that right. So, while the processor is executing the kernel mode a timer interrupt can come in which case the things will not be like this right. What happens if the timer interrupt comes in the kernel mode, you do not need to push you do not need to switch the stack and you do not need to push these extra values ss and esp right.

(Refer Slide Time: 44:07)

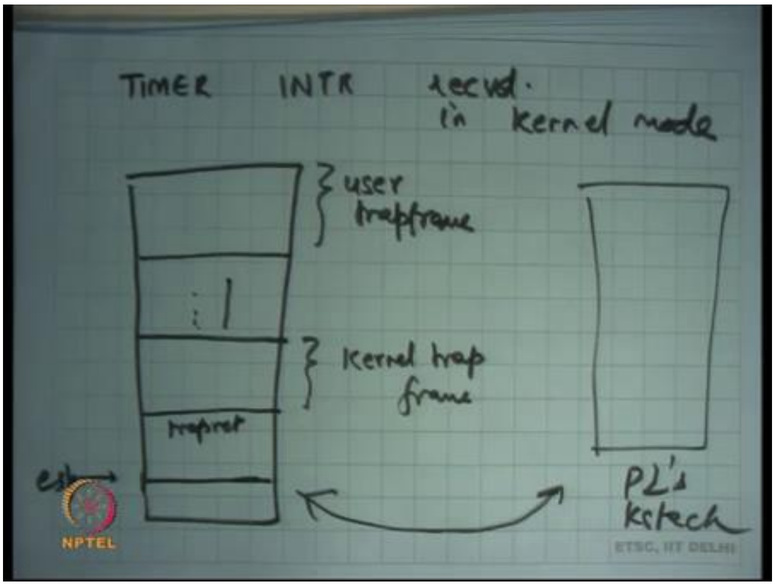


So, if you are executing, let say let say this was a kstack and let say this was a esp while you are executing in kernel mode and a timer interrupt comes let say there is a timer interrupt right. So, what will happen is, esp will just get decremented and cs, eip and eflags of the kernel at that time will get saved and the handler will get called just like before right. So, it is possible that a kernels execution gets interrupted in which case the kernel's eip gets saved right and all the other registers will also get saved just like before.

So, kernel's e x kernel is so, this time you are going to save kernels e x and all the other resistors on the stack right. So, you going to save all these things and then you may want to call the schedule function just like before right. So, the only difference between the previous case was that you started at the top and all the values that you were saving that you were saving were user's values.

In this case you do not start at the top you start at wherever you are currently and whatever values are saving now are actually kernel's values, but everything else remains the same you are still going to call the scheduler. The scheduler may again attempt to do a context switch, but that is ok, because what will happen is a context switch will happen and this kernel stack will actually have two trap frames in it right.

(Refer Slide Time: 45:41)



So, if a timer let say timer interrupt received in kernel mode and let us say you context switched then what will happen is, this was P1 kstack, you will have user trap frame after all you know if it was executing in kernel mode it mean must have come from the user at some point. So, the user trap frame must be there, some function call chain and then it got trapped again and so, this will be the kernel trap frame right.

And, at this point let us say it decides to context switch. So, from here you just context switch is to another process P2's kstack just like before if ever it comes back it is going to unwind the stack once again the kernels trap frame is going to have a trapret here. So, it is going to return to the trapret function it is going to return from the kernel trap and resume execution at exactly the same point where it was interrupted in the kernel right.

And, now it will execute again in the kernel even after returning from the trap and then it will again do trapret and then go back to the user right. So, it is possible for the kernel stack to have two trap frames right, one for the user and one while it was executing the kernel it got interrupted again, let us say let us stop.