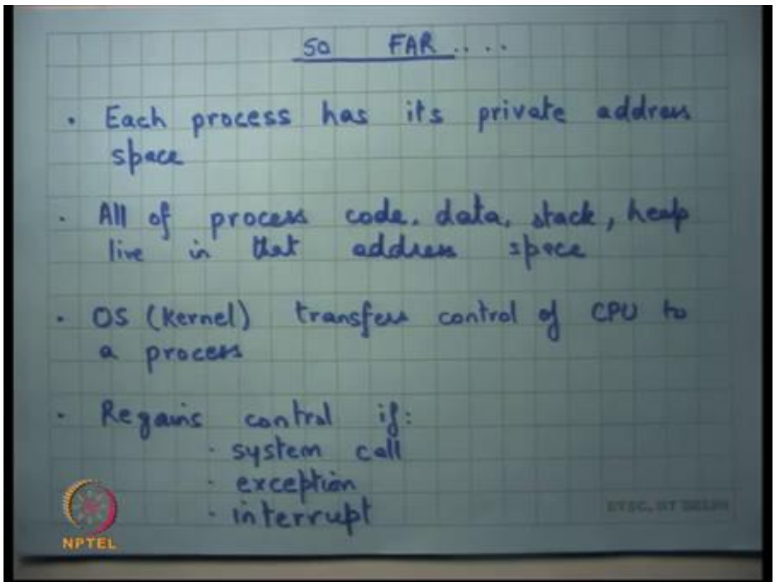
Operating Systems

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

Lecture – 09
Kernal memory data structures, Memory Management

Welcome to Operating Systems, lecture 9 right.

(Refer Slide Time: 00:29)



SO FAR . . .

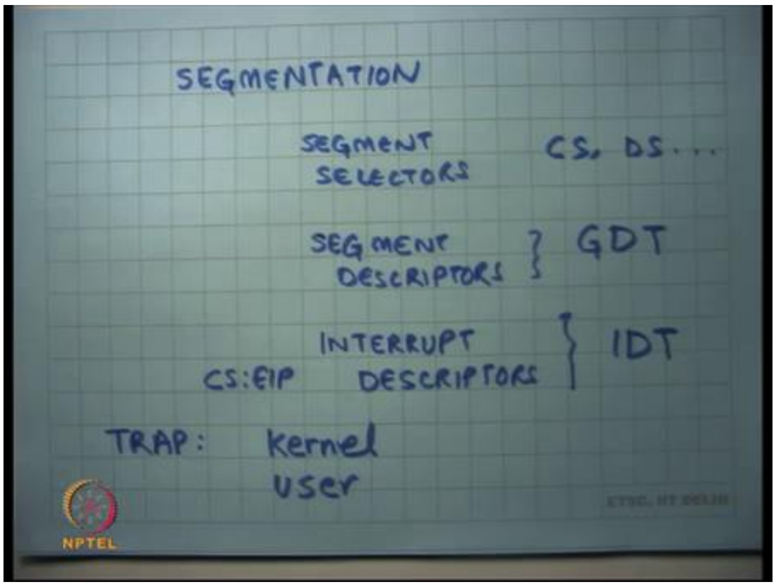
- Each process has its private address space
- All of process code, data, stack, heap live in that address space
- OS (Kernel) transfers control of CPU to a process
- Regains control if:
  - system call
  - exception
  - interrupt

So far, we have seen operating system abstractions typical operating system abstractions, a process has it is own private address space, all of the processes code, data, stack and heap live in that address space right. And, depending on what are the values of certain resistors like eip, esp etcetera you know where the code is where the stack is etcetera what to execute next.

The OS transfers control of the CPU to a process and then the process is free to run right. So, the OS is completely out of the picture when the processes running in some sense. The OS can regain control if the process makes the system call for example, right.

It causes an exception like a divide by 0 or a segmentation fault or a or some external interrupt occurs like some device needs attention in which case you know the OS comes into play right. So, we are we have we all understand it so far.

(Refer Slide Time: 01:31)



SEGMENTATION

SEGMENT SELECTORS CS, DS...

SEGMENT DESCRIPTORS } GDT

INTERRUPT DESCRIPTORS } IDT

CS:EIP

TRAP: kernel user

Then, we also said you know to implement address spaces there is a mechanism called segmentation which involves on-chip structures like segment selectors CS, DS etcetera there are segment. So, there is a global there is a descriptor table. So, there are segment descriptors whereas, and segment selectors point to segment the descriptors and they live in structures like the global descriptor table right and global descriptor table is also pointed to by an on-chip register like it is called GDTR right alright.

And, so this allows you to implement address spaces and each process can have a separate base and limit and completely non-overlapping base and limits will ensure that every process has a private address space. Also, the operating system can have a private address space for it iself and so, you know, and you should make sure that the operating systems based on limit is not overlapping with any of the process ok.

Then we said, but we also need to make implement system calls etcetera. So, you know I need to transition between address spaces and all that. So, firstly, the GDT and descriptors also have a field which says whether it can be executed in privileged mode or not. So, certain segments are allowed to be executed in accessed in only privileged mode and others are allowed to be accessed in unprivileged and privileged mode both alright.

And, then we said how do you transfer control? An application a process should not be just allowed to low say that now I want to execute in privileged mode right so, but you sometimes want to execute in privileged mode. For example, if you want to make a system call so, or there is an interrupt or there is an exception etcetera.

So, for that to facilitate that we have what are called you know interrupt descriptors and they are they live in the structure called the interrupt descriptor table right. So, if an interrupt occurs and a I am using the word interrupt to mean any of the following conditions exception, system call or external interrupt or let us we the other term for this is traps.

So, if a trap occurs the interrupt descriptor dictates how it should be handled right. So, the interrupt descriptor contains things like the co the CS colon EIP pointer and that is what you transfer to right. So, what the OS the now the so, this is the mechanism provided by the hardware. It is the responsibility of the OS to set up these interrupt descriptors properly, so that if an interrupt occurs then you know the appropriate handler gets called number 1.

Number 2, the process it iself should not be able to override the handler right and number 3 the handler itself should be very trusted piece of code it should not have any bugs or anything otherwise you know an OS a malicious process could make allow it to could actually trick it to do something which he wants with the OS should not be really doing alright.

Because if the code segment is the privileged code segment the handler will going to is going to be executing in a privileged mode alright. And, so the handler has to be very very carefully written, so that you know the process cannot ask the OS to execute his code in privileged mode for example.

And, then we said look if. So, on each of these traps there is a user. So, trap causes trap can happen trap can transition trap can happen in a kernel mode in which case you know the old CS and EIPs stored on the current stack whatever the stack pointer is because it is already executing in the kernel mode the current stack must be something that is trusted the kernel must have set it up it iself.

So, when a if a trap occurs in the kernel mode the old CS and old EIP and old EFLAGS are stored on that stack right. On the other hand, if a trap occurs in user mode it is not safe to store these values on the user stack. So, what happens is if a trap occurs in the user mode or the unprivileged mode the processor also switches the stacks right and where does how does it know where to pick up the stack from?

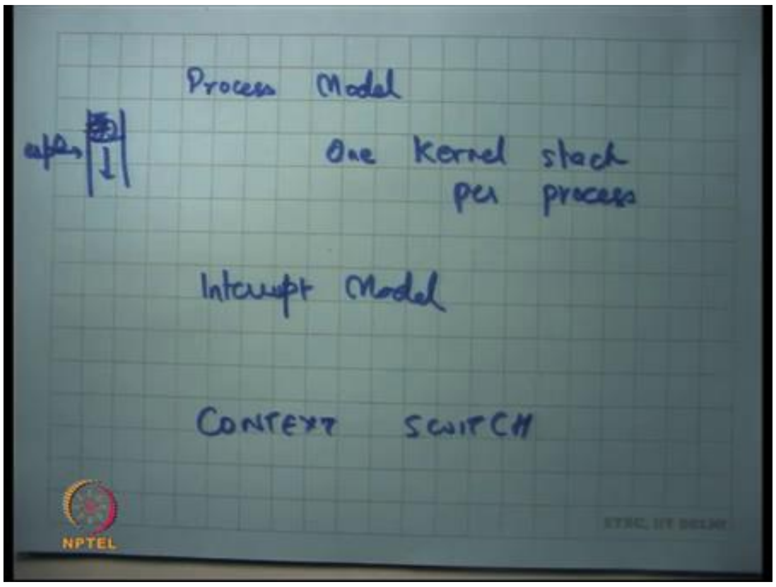
Student: (Refer Time: 05:52).

There is a data structure there is a hardware structure called a task state segment which points to the task state segment and so, you can basically pick up that so, the hardware knows here is where to pick up those stack pointer from and it loads that stack pointer and then pushes these things onto it right.

So, because it is also loading the stack pointer, it also needs to save the old stack pointer right. So, if you check take a trap in the user mode you are saving 5 values to the stack; if you are taking a trap in the kernel mode you are saving only 3 values to the stack alright. So, it is the responsibility of the kernel to set up this task state segment appropriately before transferring control to the process right because if it does not do that then if a trap occurs or a system call occurs then things can go back ok.

Now, the other thing is that how does. So, each process has an address space; each process also needs to know you know what is the kernel stack that it is run on right.

(Refer Slide Time: 06:55)



Process Model

One kernel stack per process

Interrupt Model

CONTEXT SWITCH

So, we said that the kernel could be implemented in 2 ways – one is the process model where one stack per process one kernel stack per process. In this mode whenever you transfer control to a process you have a special area which you call this process is kernel stack right special memory area and that that is basically what you load into the task state segment. So, before you actually transfer control to process P1, you load P1s case stack; before you transfer control to P2 you load P2s case stack and so on right.

In the other model which is the interrupt model you basically say let us just have one stack right and so, irrespective of whether you transfer to P1 or P2, the stack will be the same which is you know one kernel stack let us call that the k stack right. And, so basically that means, what that means, is that whenever a process actually transitions to kernel mode it will always start with a completely fresh stack right.

Completely fresh a fresh stack basically means it has nothing else except the 5 values that were pushed on the trap it iself right 5 or 3 whatever the case may be. So, you know in the interrupt model whenever you are transition from the user to kernel typically you will start with a fresh stack alright. Actually, even in the process model typically you always start with a fresh stack, but the kernel actually has the flexibility to actually you know say that look these are some values already pushed in the stack.

And, so in the process model for example, I could say here is my you know kernel here is my esp 0 and I could have already pushed some things here and the when the trap occurs actually the 5 values get pushed here and so, the stack is not completely empty. It also has other stuff which the handler may need right, but typically this is not used you also start to the fresh stack in the process mode right you do not really have things that are saved previously and similarly in the interrupt model.

Of course, it is possible that while I while a process was actually executing in the kernel mode. So, you know so now, I am going to use this terminology a process could be executing in user mode which means the process is executing in it is own address space with it is own eip and all that or the process could be executing in kernel mode.

What I mean by a process executing in kernel mode is let us say a process executes at system call, I transitioned to the kernel it is not the process which is executing, but it is the kernel which is executing on behalf of the process alright. Or you know there is an interrupt that happened. So, it is a kernel there is an external interrupt that happened, so, it is a kernel actually takes over and the kernel is executing on the stack that was you know for example, the stack of the process that was originally running right.

So, even in this case I am going to say that the process is running in the kernel mode. It is really the kernel running on behalf of the process or in the in other words is the kernel running on the stack of the process in the process model right ok.

Student: Sir, interrupt model if there is something already in the kernel stack then it will delete that first?

Right. So, I am explaining how it works in the interrupt mode. So, let us say a process was executing in the kernel mode and now the process wants to switch the stack alright or switch the start another process running.

So, let us say a process made a system call like read right and the file descriptor of the read was actually pointing to a file which lives on disk and you as you know disc has actually a very slow device it is going to take a lot of time right. So, a process said read the kernel figured out that it is going to it wants to read from the disk, now one option is that the kernel could just you know the you know keeps spinning waiting for the disk to complete you know that is a polling model.

The other way to do it is you say you know I know that disc is going to take a lot of time. So, I basically say I want to switch the CPU right. So, I am going to get the CPU from P1 to P2 while P1s read system call is actually executing on the disc. So, P2 can execute on the CPU while P1 is executing on the disc so to say right.

Now, when I am doing this when I go when the disc actually completes and I want to get back to P1, then I should know exactly where to start from. In this case, I was I should start from within the kernel execution right because I was interrupted within the kernel execution and I switched to P2 and now I want to cut back to P1 in the kernel mode and so, the kernel stack should be exactly has as it was left before the context switch right. So, the switching from P1 to P2 is also called a context switch alright.

And, what you need to do in the interrupt model is that when you do a context switch you also store the contents of the stack somewhere where they are from where they can be restored when the P1 gets context switch back right or when it gets stored right.

In the process model you do not need to save the contents of the stack you could just save the stack pointer and that is enough right. So, that is the; that is the real difference, there is no fundamental difference. In one case you are storing the contents of the stack if there is a context switch that occurs within the kernel execution, in the other case you are just switching the stack it iself alright.

So, that is our review of last times discussion. So, I received some questions over email and there was also a question last time that can not a kernel just jump to the user mode directly using let us say the l jump instruction instead of the you know instead of the iret mechanism that we saw last time.

Recall that we saw this mechanism where an external interrupt or an exception or a software interrupt enters the kernel after pushing some data on stack and then there is this iret instruction that pops the data of the stack and then returns back to the user typically right.

And, so I said even the first process typical I mean this stack could either be created by an entry to the kernel by the real entry to the kernel by hardware or it can be manufactured by the OS kernel it iself and simulate it, an iret is simulated as though an interrupt return has happened although it has not happened to transfer control back to the user.

And, so I think the question was why do I need to do this sort of complicated mechanism where first where first I setup the stack and then I call iret, why cannot I just do l jump to that particular code segment of the user? Well, it turns out that ljump does not allow you know change transitions across privilege levels ljump only says selector alright. So, nothing fundamental, but just a matter of fact.

The other question I got was over e-mail was that when you context switch to 2 processes I said you change the address space right. So, I said that you know let us say the first sort of a dumb way of switching that space is you change the GDT every time you change the process.

So, you switch from P1 to P2 you also reload the GDTR each time you do that right before you cut transfer control to, but we said you know that is too wasteful. You know GDT is actually 2 to the power 13 which is pretty big, and you know if I want to have one GDT per process that is too wasteful.

The other way to do that was let us you know override the values of the GDT it iself each time I context switch right. So, each time I context switch I override the values of the GDT and then I can go back. And, so the question was why do I need to even override the value of GDT? Why cannot I just have you know that 2 to the power 13 entries in the GDT?

And, I am so far, I have just said you know there is you know 6 segments and so, you know what the other entries are with really, therefore. Why cannot I just say you know process P1 these entries, process P2 these entries, process P 3 these entries.

And, so when I context switch, I do not even override the GDT I just reload the segment selectors and I am done. So, my GDT contains the segment selectors of all the different processes. When I want to context switch from one process to another, I just change the segment selector, what is the problem?

Student: We can actually change further process.

Yes. A process is free to change it is segment selector right. So, if the GDT has segment descriptors of other processes then a process can easily just change override a segment selector to point to somebody else's address space. And, so that is a security risk P1 can now access P2s memory which was not supposed to happen right. So, you actually need to overwrite the GDT to do with that. Sure, question.

Student: (Refer Time: 15:03) to ensure that it does not accept another processes address space?

Sorry.

Student: We use the limit to the there is a (Refer Time: 15:15) limit.

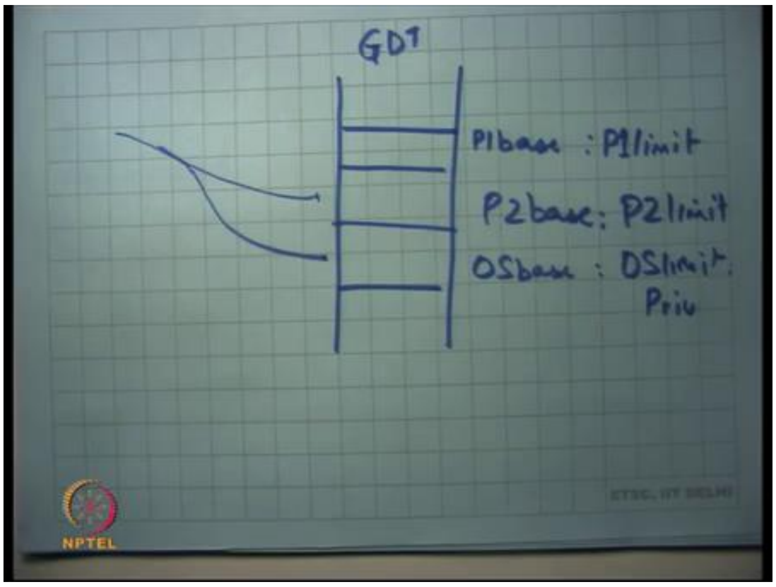
That is in the descriptor right.

Student: Yes.

So, ok.

Student: So, while we are changing the CS to point to some other process it finally gets converted to a physical address which is of some other process. So, at that time we are checking the limit also (Refer Time: 15:31).

(Refer Slide Time: 15:39)



So, let us see how it works right. Let us say this is my GDT right and let us say this was P1s P1 base colon P1 limit right and let us say this is P2 base colon P2 limit alright, and let us say this is OS base colon OS limit – this is the kernels address space right alright.

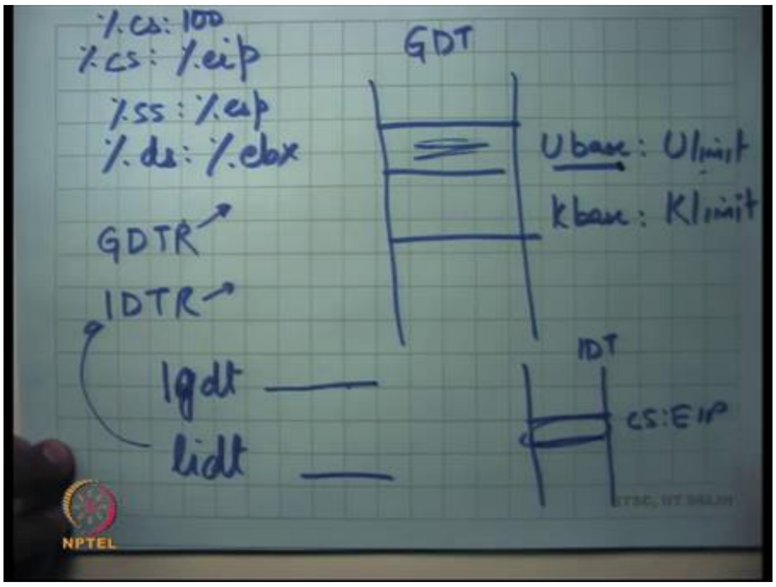
Now, when I context switch if I just switch the segment selectors you know, and a process is allowed to overwrite it is own segment selectors. So, what it all the process need to do is point to this segment selector right. So, P1 can now point to P2 and so, that is a problem right. P 2 cannot P1 cannot still point to OS because you know the OS will have a bit which says privileged right.

So, you cannot in an unprivileged mode point to the OS is a address descriptor, but you can point to another process as descriptor because that is also unprivileged right and so, the you do not want to allow that. So, each time you switch between 2 processes you actually you know ensure that the P the descriptor table GDT has only entries which this process should be allowed to access.

Student: So, technically that there is no entry of P2 and GDT on P1 is (Refer Time: 16:58).

Yes. So, there should not be any entry of P2 when P1 is running in the GDT ok. So, this is you know this is the; this is the way that that was a suggestion in my in the question in the e-mail, but this is not how it is done right. So, you would not have separate descriptors of P1 and P2.

(Refer Slide Time: 17:17)



What you will have is you will have a GDT right and you will have something called user base and user limit and you will have kernel base or OS base and kernel limit right. So, you will have just one descriptor for the process and each time you context switch you want to overwrite this descriptor alright that may you know that may bring you to the question you know why do I need 2 to the power 13 entries, you know if that is all I am using the GDT for.

Well, you know other way is to use segmentation hardware, but this is the typical way an OS will use that. For example, you know this is how Linux would use it or Windows would use or you know the your programming assignment OS Pi OS is going to use it or what we are going to study later xv6 is going to use it. So, most operating systems use this hardware in this way you know.

So, a hardware designers imagine a certain way of way usage, the operating system designers figure out that you know this is the best way to use it and so, you know there is some extra things that go into the hardware because the hardware was designed before the OS is written.

And, you know even the ring levels are an example of that right. The hardware designers actually designed 4 ring levels 0123 and the intention there was that you know you will typically need multiple privileged levels you know you will need 0 for the kernel, 1 for the device drivers you know this is typically what you hear, then 2 is for let us say the libraries the system libraries and then 3 is for the real application right.

So, that was the; that was the imagination of the hardware designer. The OS designer said no I do not need all this that is too complex I just need 2 privileged and unprivileged and so, I am just going to use 0 and 3 alright.

So, the other question I got was you know you know I refer to the Wikipedia page of GDT and the GDT has also other things like task state segment, LDT local descriptor table, call gate what are these for. I would say ignore it you know. We do not need it for this course. Once again there are features that are present in the hardware, we do not I mean there is so many features that we do not we are not talking about alright.

There was another question – sometimes an exception is customized according to a process. So, question is you know can a process actually customize the exception handler according to itself. So, saying that you know handle my page fault in this way or handle my segmentation fault in this way, handle my divide by 0 error in this way right or handle the interrupt from this device in this way and so on.

Can a process can do that? Well, in the instructions we have studied so far that is not allowed right. What executes in the kernel on an exception is not controllable by the process. What is controllable by the process is signal handlers.

So, you know for example, Unix provides you this mechanism of converting a hardware interrupts or exceptions to software signals right. So, if for example, you actually make data divide by 0, the hardware is going to generate a floating-point exception. The OS is handler for that floating-point exception is going to get called. The OS may convert it into a floating-point signal and give it to the process which actually executed that offending instruction.

And, so basically what will happen is that the process which was executing will now straight away jump to the signal handler of the FPE, the floating-point exception right. And, so the signal handler of the floating-point exception can be customized according to the process ok. So, the process can customize the handling of at the user level handling. We cannot customize the process the kernel level handling of a certain condition alright.

Student: Sir.

Yes.

Student: Sir, but how would I tell the interrupt handler to generate that signal?

How will you tell? So, you know Unix basically has a standard so, have you known certain semantics. For example, a floating-point exception will always generate the floating-point exception signal, or a segmentation fault will always generate the SigSev the segment. So, segmentation exception on the hardware will always generate a segmentation exception in the signal in the software to the process alright and the other ways by which segmentation signal can be generated and so on.

So, there are certain semantics that always an exception at the hardware will generate a signal in the process level. And, now the process and also typically the process will have default handlers for these signals and the default handler will just say you know we will just kill it kill you. There are certain signals that can have that can be overridden the handlers of which can be overridden and there are certain signals to handlers to which cannot be overridden for security reasons.

Student: Sir. So, suppose means there is an interrupt handler and there is a signal handler and in the interrupt handler I generate a signal.

Hm.

Student: Now, as a set of signals both in my interrupt handler and in my signal handler.

Ok.

Student: So, these will get executed first.

The question is there is an interrupt handler and there is a signal handler; the interrupt handler lives in the OS the signal handler lives in the process and interrupt that occurs actually first in so, you know what happens both execute, does one executed etcetera. So, the answer is that they both execute right. So, when the interrupt occurs the kernels interrupt handler gets called, the kernel interrupt handler is the one which is actually going to generate the signal.

What does generation of a signal mean? It just means that it is going to override the instruction pointer of the process with the value of the handler, that is all. And, now it is going to transfer control to the process just like you know it was doing earlier, but the only thing is the process is going to feel that it is suddenly jumped from where it was to some somewhere else. So, that is what generating a signal means. Question.

Student: Sir, why is there a need of these 2 abstractions interrupt and signal if all that the interrupt does is generate a signal?

Not always, right. So, the question is why do we need 2 obstructions interrupt and signal?

Student: Ha.

If all that an interrupt does is generate a signal.

Student: Yeah.

So, an interrupt. Firstly, an interrupt does not always generate a signal alright. An interrupt could be coming from a hardware device like a disk and in which case your device driver just needs to just copy the value that is sort by the disk to some buffer in which case you do not did not need to generate the signal. So, you if the OS generate is operating the different environment right, the process in operating in a different environment; the question you are asking is why do I need signals when I have interrupts.

The reason is because interrupts cannot be passed directly to the process because interrupt handling needs to be privileged mode handling because you are dealing with real devices, real resources right. On the other hand, you know you need signals because the process also needs similar kind of event driven behavior that is something has happened you know I want to know about it and I want to know it in an interrupt driven fashion as opposed to keep checking.

So, you know you could you know Unix signals as not a necessary abstraction I could just say you know I do not need you know my operating system does not support signals you know. If you have made if you if some error condition occurs if you make an exception I will always kill you alright and if you for example, make some kind of inter process communication or some kind of you know device communication then it is a responsibility of the process to keep checking rather than a signal based mechanism.

So, you know this is also possible abstraction in which case an interrupt will not be converted to a signal right. So, just want to when you are designing an abstraction you basically want to look at you know what the common things are and what happens. So, and basically what it turns out is that the abstractions that the hardware designers implemented those are the and the abstractions that the operating system designers implemented, there is some similarity between them.

Signals are an instruction that the operating system designers implemented; interrupts are an abstraction that the hardware designers implemented alright. So, there was another question let us say I have a USB port and there are multiple devices connected on the USB port you know you can connect a mouse and a keyboard to the same USB port using a hub or whatever. And, so now you know there is an interrupt then how does the OS know whether it is a keyboard handler that I should call or the mouse handler that I should call?

Well, an interrupt does not just it just says that this part all the devices connected to this particular port or you know all the devices for which the interrupt vector is this one of those devices has required my attention. So, that is what an interrupt means.

Now, the interrupt handler will typically now look at all the devices, I trade over them and check, ask all of them – do you need my attention, do you need my attention and so on and if whoever says I need your attention he is going to give the attention and it is going to say what attention do you need right.

So, it is interrupt is just a way of say telling the processor or in that I need attention. It is like you know a phone call versus an e-mail right. A phone call just immediately gives you some attention. You know when you get a phone call then you ask you know who you are etcetera what do you need etcetera and e-mail is basically it is in the e-mail box that is a polling mode in which case you going to just check on your own.

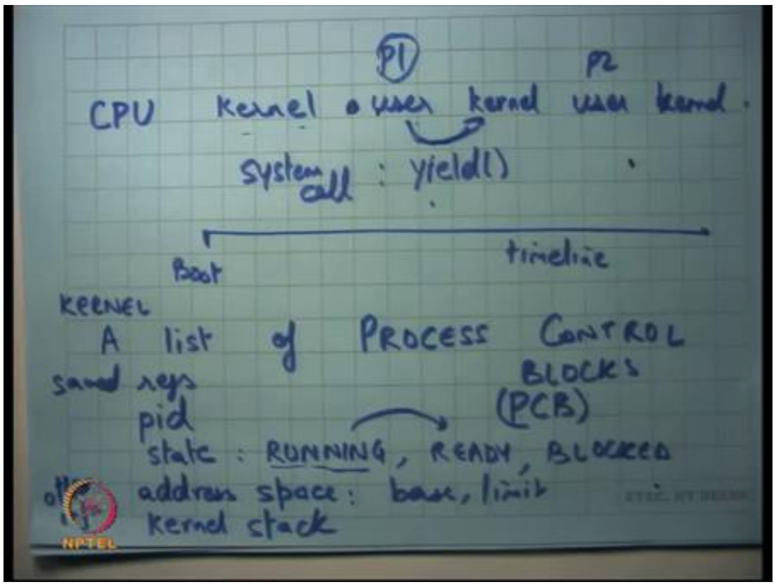
In both case the operation is the same is just the way of notification that is different alright. Finally, I want to I said that there are 2 pointers on the chip. There is the global descriptor table resistor which is the which is points to GDT and then there is the interrupt descriptor table resistor. So, my question to you is the GDTR the value in the GDTR is that a physical address or a virtual address.

So, what is the virtual address? Virtual in segmentation is of the form of you know percentage some selector like you know cs colon percentage eip – this is a virtual address or percentage ss colon percentage esp or you know ds ebx and so on. These are all virtual addresses right. Or I could say cs colon 100 basically means offset 100 in the segment called cs right. So, GDTR what does it need to be? can it be a virtual address, or it does it need to be a physical address?

Student: Physical address.

Physical address right because I mean the translation from virtual to physical needs to go through this trans this GDTR and so, if that is a virtual address then you are in an infinite loop. So, that is why it does not make sense. In fact, the IDTR also is not a virtual address it is a physical address and if you in the terms of in what we are saying you know when we were only using segmentation alright ok.

(Refer Slide Time: 27:51)



P1 P2

CPU Kernel user kernel user kernel

system call : yield()

Boot timeline

KERNEL

A list of PROCESS CONTROL BLOCKS (PCB)

saved regs

pid

state : RUNNING, READY, BLOCKED

address space : base, limit

kernel stack

So, we saw that the process the so, basically so far, we have seen that the kernel executes transfers control to the process. So, if I am a CPU you know, and this is my timeline.

Student: Sir.

Give me a second ok. Question?

Student: Sir, how does the (Refer Time: 28:00).

How does the?

Student: (Refer Time: 28:02).

Sorry, how does the what fit in?

Student: How does the other physical address fit in the GDTR?

How does the physical address fit in the GDTR?

Student: (Refer Time: 28:15).

Using the lgdt instruction, right. So, lgdt instruction allows you to set up this value ok. So, it is the whatever the argument to this gets filled into gdtr operand to the lgdt instruction. Similarly, lidt instruction fills in IDTR load GDT load idt that is what it means alright. More questions in the GDT descriptor the U base, is it a physical address or a virtual address?

Student: Physical address.

Physical, right I mean it does not make sense it is a physical address. Limit, it is just a so, it is.

Student: (Refer Time: 29:00).

It is just a value it does not matter. What about the IDT descriptor? IDT what does it have? It has a physical address or a virtual address?

Student: Virtual address, virtual.

Virtual. It is basically CS colon EIP right. So, it is basically specifying both the segment and the offsets. So, it is a virtual address right. So, if I look at the CPU and I look at this timeline basically what happens is let us say this is boot. So, the kernel executes for some time where it boots up and then it creates some process internally. So, it creates some address space for example, it creates a segment for the address space and now, it will set up the stack and call iret and here is the first process that gets run. So, it starts executing in user mode.

Now, this process is going to make some system calls or you know some external interrupt occurs and so, some other kernel activities going to happen and then it is going to go return back to the user then it happens kernel and so on right. So, that is the timeline of a CPU. Sometimes it is executing in kernel mode, sometimes it is executing in user mode and etcetera.

Now, in the when it is executing in the user mode it could be executing let us say P1 earlier now it is executing P2 and now it is executing P1 again and so on right. And, each time the kernel it switches back to kernel mode the kernel has a choice whom to get to run next and the sets it sets things up accordingly and transfers control to that particular process. So, (Refer Time: 30:27) sets up the sets up the kernel stack, it sets up the address space and then transfers control to that particular process.

So, to be able to do this it need certain data structures. For example, it needs you know it needs let us say the typically it will have. So, a kernel will store a list of process control blocks or PCBs right. So, this is basically all the active processes in the system at any time. Let us say this is boot time. At boot time there is no active process in the system. In fact, during all this duration there is no active process in the system.

At this point the kernel what it does is it adds, it sort of creates manufactures a process there was no process really at that point the manufacturers the process processes it means that manufacturers, it creates a new address space, allocates the new address space, creates process corresponding base and limit entries creates a kernel stack and adds this process P1s value to this list of PCBs. So, adds the PCB to this list right.

What does the PCB triple typically contain? It will contain the idea of the process pid alright. It will contain things like state. What is a state? Whether it is currently running whether it is ready to run, but it is not currently running. So, things like you know whether it is running a process could be actually running right now right or whether it is ready to run, but not running.

So, let us call it ready or let us say it is blocked you know; blocked is basically things like I made a request to the disc and you know I am blocked. So, I am not ready I am not eligible to run on the CPU till the disk device comes right. So, what will happen is the process that made the read system call that is put into the block state the device driver the devices activate a you know informed that you please give me the value of these disk blocks, the device is going to take some time and after that the device actually invokes an interrupt. The kernels interrupt handler come gets executed.

The kernel interrupt handler figures out you know which request has actually finished and it figures out that the require that was finished is actually the request that was made by this particular process and so, now at this point it will transition it from block to ready alright.

So, that state now there are other things like address space you know for example, what are my base and limits alright and segmentation that is the address space that is all you need to store. You know let us say the kernel stack, if I am using the process model then I store only the pointer or if I am using the interrupt model then I also then I store the contents if any and let us say other information you know, this is just example. We want to look at what kind of information may be stored, but this is what a PCB store right.

Where are where is this list stored? In the kernels address space of course, right. You do not want the process to be able to ever read or write to this list this is completely private to the kernel and you are implementing this using the kernels segmentation the segmentation at the base and limit of the kernel alright.

Now, the question is how does this transfer in what conditions does this transfer happen? We have already seen it can happen to a system calls, exception, or interrupt. Now one of the system calls happens to be yield right. So, you know so, for example, Unix will have a system called yield and the semantics of the yield system call is that you know the process figures out that I am actually waiting for something and he wants to say that you know I am waiting for something and I do not want to take up the CPU.

So, it is like being a good citizen and he just says I want to yield the CPU. So, he just says I want to yield the CPU the kernel you transition to the kernel. So, the way it says he wants to yield the CPU there is a system call called yield that is only way you a process could communicate with the kernel. So, the kernel the process says yield, the kernel gets to run, the kernel figures out whether there are other processes who are also waiting for the CPU and if so, it schedules another process to that CPU right.

If there are no other processes then it just says you know you are trying to be good, but you know there is nobody else, why do not you just take it alright ok. So, that is a yield system call. So, that is one way that you could transition from user to kernel in which case you basically transition the state of the process from running to ready right.

So, if actually you know if somebody called yield you basically just transition his state from running to ready and you transition another process state from ready to running alright. But you know not all processes are good citizens. So, there could be a process which just a you know never calls the yield.

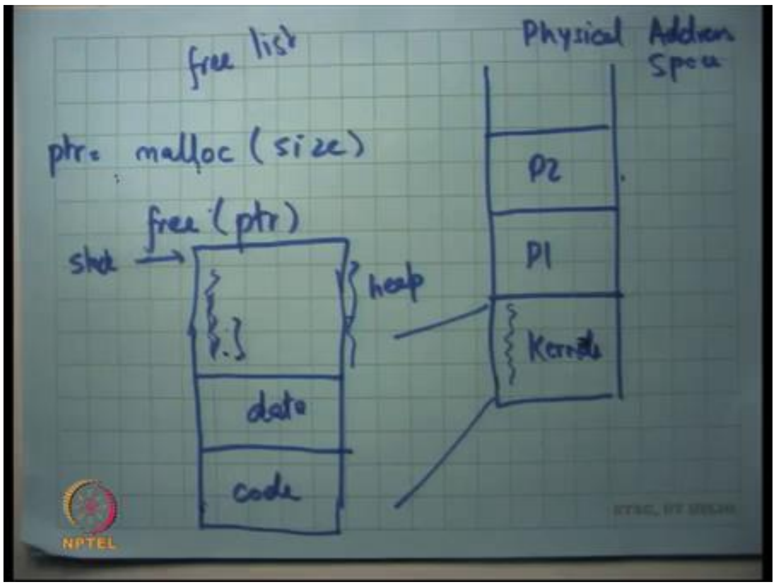
So, what do you do? So, we have discussed this before the kernel has you know has heavy unridden mechanisms in which case it basically sets up the hardware device called timer. So, every you know every computer system will have a device called the timer device on it is motherboard let us say and so, that device is what the kernel has configured before transferring control to the first process.

So, the kernel has configured the timer device to say periodically generate an interrupt 100 milliseconds right. So, even if the process is not calling yield 100 milliseconds the kernels timer interrupt handler will get called right. On the timer interrupt handler may do this operation even if the process is not doing it right. So, a process can do, call yield or the kernel can do it on his behalf. In either case you are sure that a process cannot run away with the CPU alright ok.

So, this mechanism of shifting from one process to another which is converting my state from running to ready and converting somebody else’s state from ready to running it is called a context switch right and of course, it is not just switching state. It also means saving my data which means my resistors, my stack in the PCB right. So, one of the things that the PCB also contains is let us say resistors right save resistors.

So, all these things are saved in the PCB – the state is modified from RUNNING to READY, another processes state is modifying from READY to RUNNING, its registers are loaded from it is PCB into the hardware state. Stack is set up properly and you give control right and eventually it will be going to call iret which is going to go back to the process alright ok. So, this is the timeline of how the kernel executes alright.

(Refer Slide Time: 37:23)



So, far we have seen that you know let us say this is my physical address space. And, the OS segments it into you know kernels area, P1s area, P2s area and so on, right. Now, and the kernel internally is storing things like the global descriptor table, the interrupt descriptor table, the list of PCBs and other data structures that it needs and also the handlers all the code that is going inside the handlers it is all living in this region right.

Now, let us look at something which is more fine-grained question is how the kernel manages the space right. So, this is just a chunk of space that the kernel has reserved for it iself, now how does the kernel manage the space? So, typically the how a software manages space using these functions called malloc which is memory alloc, and free which is memory free, right.

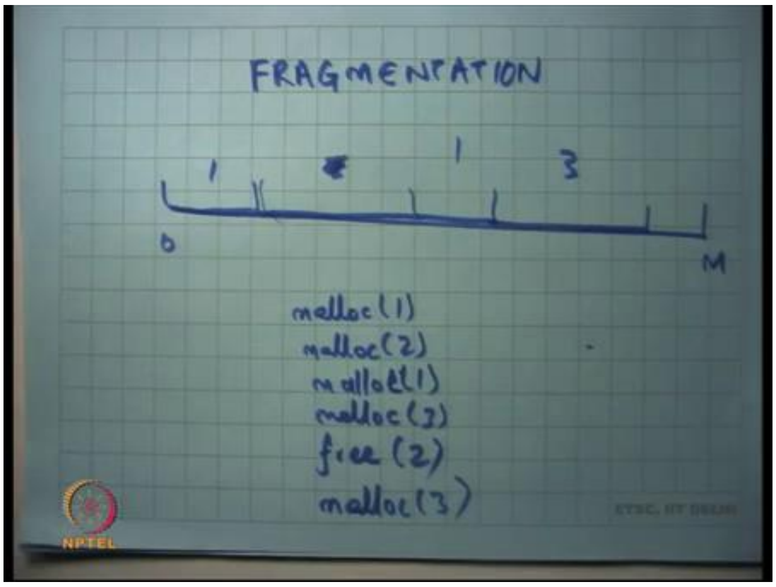
And, so what malloc takes is a size and returns a pointer and what free does is, it free is that point. I am sure all of you have used malloc and free yes alright. So, and of course, malloc can fail right. So, malloc may return the null pointer in which case the malloc could not happen and this could happen if you know you have been mallocing too much and you actually run out of the space that the kernel allocated for it iself right.

So, let us see how malloc would be implemented. Typically, the way it is implemented is let us say this is the kernel. So, I am just you know magnifying this area let us say this is the kernel. So, the kernel it iself will have some code, it will have some data and it will initialize its own stack somewhere here right. So, these are all the stacks, that is it. So, let us say it is an interrupt driven model let us say initialize the stack here.

So, there is only one stack and it initializes here right and now everything else which remains in the space is what is called the heap which is managed by these functions called malloc and free. Initially you just add the entire space to a list called a free list right. I mean add the entire space to the free list. You reserve certain space for the stack and all the other space you basically say let us call it in the free list.

Each time a function calls malloc each time the kernel some part of the kernel calls malloc, you are going to look into free list get that amount of chunk of memory and return it right and so on. Each time somebody calls free you are going to add it back to the free list right. So, you can maintain a list of free memory locations.

(Refer Slide Time: 40:21)



So, let us see how this works. Let us say this is a address space, that is the heap 0 and to you know some value M and let us say you malloc 1, then you malloc 2, then you malloc

1 again then you malloc 3 and then you said free 2 right. So, let us say I said malloc 1, malloc 2, malloc 1 again, let us say malloc 3 and let us say my memory allocator is just doing contiguous allocations, just increments a pointer and just gives you the next available memory location and then I say free 2 right.

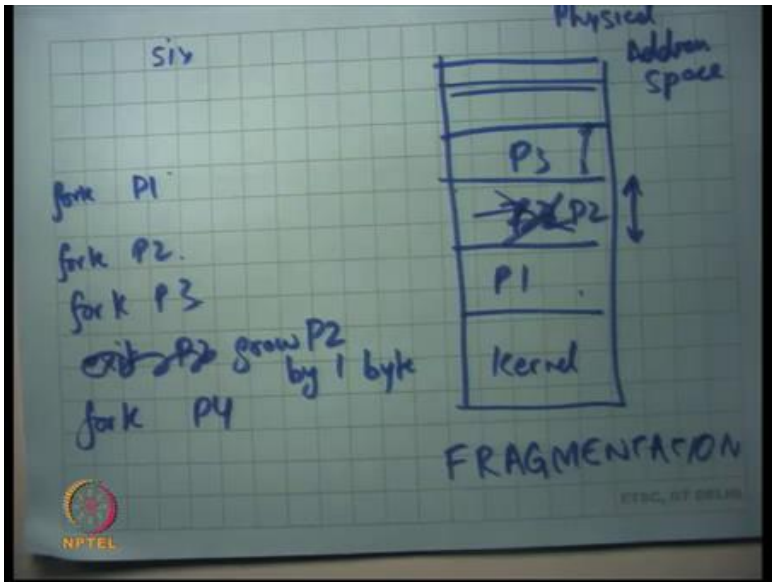
So, it is going to happen is this area is going to get freed and it is going to get added in the free list and then I say malloc 3 again and so, what happens is this is a region of length 2, you have asked for 3 bytes and I cannot just give you these right. So, what because there is no contiguous here and so, what I have to do is I have to go here.

And, so what going to happen eventually is that there can be lots of small holes in your address space right. So, you have areas that have been malloced and then there are holes in the middle that we freed, but the holes are actually not big enough to satisfy many of your requests. So, eventually you are your address space will get what is called fragmented right. This is called fragmentation.

So, there are many algorithms on choosing where to you know when somebody calls malloc the function malloc has freedom in choosing exactly which area to give right, and exactly how to manage the space in such a way that you minimize fragmentation is something that we are going to look at later, but we should at least know what the problem is ok. Same thing exists in the process level.

So, you know I motivated fragmentation in the context of the kernel, but similarly the process has this very similar structure there is code, there is data, you know so, there is somewhere there is a stack and now you have the heap everywhere right and, now the function the process internally is going to use manage it is own address space using malloc and free. It just manages heap using malloc and free alright.

(Refer Slide Time: 42:31)



But, when I look at the physical address space again, so, let us say this is the physical address space and these are the kernel this is P1, this is P2 and so on. And, so in a typical execution of the kernel you are going to see lots of process creations and process exits right. So, process is going to get created, you are going to allocate space for it, and you know you are going to.

So, let us say you know P1 got forked fork P1 and then fork P2, then fork P3, then you said exit P1, exit P2, let us say you said fork P4 right. Once again, we have the similar problem. So, you know P1, P2, P3 got created like this, then P2 got P2 exited and now I want to allocate space for P4 let us say P4 was bigger than P2 you know I have a hole that I cannot use and I have to basically use extra memory. So, there is a fragmentation in the physical address space.

What are some possible solutions, right? So, let us say I wanted to look for P4 and let us say I know searched in I address space and I could not find a block that is big enough for P4. One thing that I can just say you know fork succeed did not succeed. I return a negative value to the fork system call, that is one way, but you know what if you know the sum of all the holes is bigger than P4, then can I do something smarter?

Student: Shift.

I can just shift P3 down. What does it mean to shift P3 down? I just change the base and limit appropriately and the process will never know, right. So, this indirection which is the segmentation hardware is actually allowing you to move the address space at will inside and without the process ever knowing what you just changed the basic limit you move the P P3 down. The process code has nothing to do with it right because it sees still the same addresses 0 to whatever maximum it is allocated alright ok.

But, you know it is not a very convincing solution because a process may be large and copying between memory is an expensive operation because you know there is lots of bytes that are going to go over the system bus you are going to and so, because processes are big copying the entire processes is an expensive proposition.

Also, you know let us say I did P1, P2. So, let us say I do fork P1 fork P2 fork P3 and then I just wanted to say grow P1, grow P2 by 1 byte and no, not possible because there is P1 you know there is P3 right above P2 and so, if I want to grow P1 by 1 byte actually I have to copy I have to find you know that space and then I have to copy the entire P2 there and then I grow it by 1 byte. Once again it is very efficient way of doing things.

Student: Sir, but we are only changing the base value we are not copying the entire (Refer Time: 45:31) physical memory.

Before you know you change the base value, but you also copy the physical memory contains right, you cannot just change the base value right ok.

So, these are problems which are again you know come what are called fragmentation problems. And, the reason they occur they exists is basically because the segmentation hardware allows us to only allocate contiguous regions of physical address space and give them to the memory process. So, I cannot say you know P2 as to lives here and here and here.

So, I cannot have lots of different you know I cannot just say P these are P2s areas and I cannot just have the list of P2s areas. I need to say that P2 lives in this contiguous address space. If I were able to say that P2 lives you know scattered around in the memory, then it will be much easier I could just do you know do things in a better way.

There is some respite though because as we said you know segment there are 6 segments. So, I could potentially have 6 different contiguous regions, but now the programmer needs to be careful about you know which segment it is it does not see a uniform address space. It actually segmented address spaces this the code segment, this is the stack segment, this is the data segment it is possible.

But, you know typically you do not use it complicates the program the compiler it program it complicates the job of the programmer or compiler job of the compiler writer and so, it is not typically done. You would like it is much easier to actually write programs for a uniform address space as opposed to a segmented address space alright, but that is yeah question. So, I think there was a question there.

Student: Sir, you said that when the OS allocates the memory for a process, it does not know beforehand how much memory the process needs.

Right.

Student: So, should not it do something like the OS should allocate equal mb to all the processes?

Question.

Student: (Refer Time: 47:31).

Right. So, the question is you know an OS does not know a priori how much memory and a process needs right and so, question is should not I allocate equal memory to each process? Yes, I mean that is one way of doing it I mean see what are the abstractions? The abstractions are fork and exec and think like that right and then we have malloc and all that right.

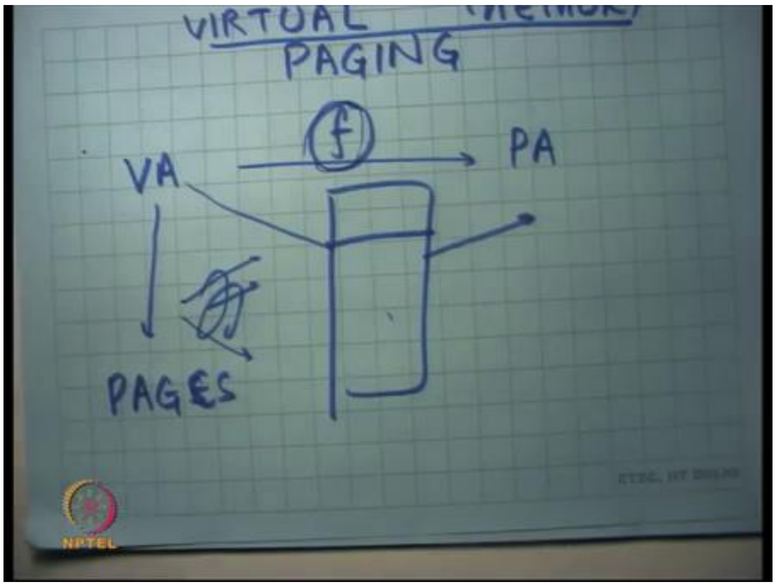
So, when I do fork, I have no idea what this process is going to actually execute. So, I just you know I just create an address space and you know in segmentation what I will do is I will probably create the address space is equal to the parents address space, but let us just exec an executable that requires much bigger memory.

Student: But parent address space cannot increase. You said that if a process tries.

Parents address space will not increase, but the child's address space may increase right because you called exec and the executable that you are going to load now is much bigger than the parents address space ok. So, I mean basically the point is that you need dynamic growth of processes and this way of doing address translation is not supporting very fast way of doing dynamic growth of processes. Question?

Student: You answered that.

(Refer Slide Time: 48:45)



So, we going to look at next lecture another mechanism called paging alright. So, right now so, basically, we are we really talking about a mechanism of in the operating system called virtual memory. And, so what is virtual memory? Virtual memory translates an address which is called the virtual address and to a physical address and there is some function f which does this translation.

So far in segmentation this f was a very simple function where we just said you know VA is equal to base plus PA is equal to VA base plus VA and you also check it against the limit. But what you really need is that this function should be more sort of detailed you could be able to say this byte goes here and this byte goes here, and this byte goes here.

And, so what does what is the most general function? What is the most general function is for each byte basically have something which says you know this is where you should this is where this byte lives this is where this byte lives, but the at the data structure if you to store this mapping will be bigger than address space it iself right.

So, what you basically do is you basically divide the virtual address space into what are called pages and then you have a table in the function which maps one page in the virtual address space to a another page address space in the physical address space and we will go and discuss more details in our next lecture alright ok.

So, let us stop.