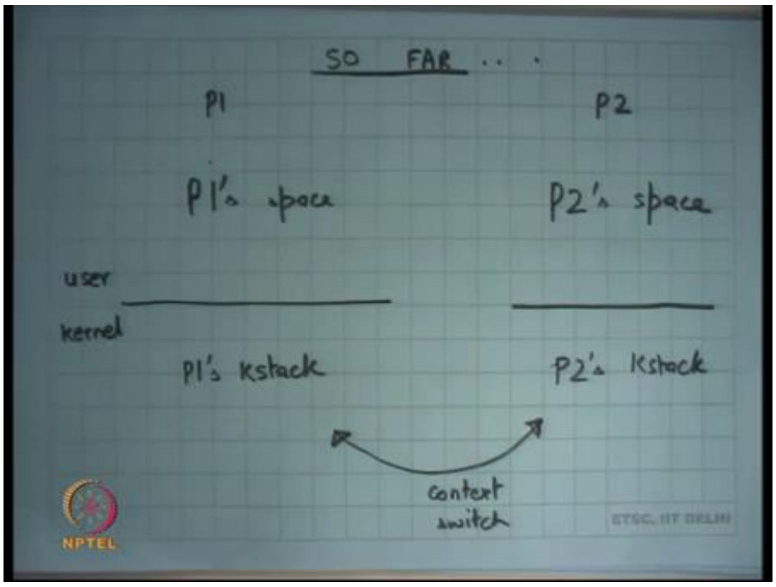
Operating Systems

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

Lecture - 18
Process kernel stack, Scheduler, Fork, Context-Switch, Process Control Block, Trap Entry and Return

Welcome to Operating Systems lecture-18.

(Refer Slide Time: 00:27)



So far, we were discussing how processes are implemented and how they work, how they work in action. So, let us say there are two process is P1 and P2. P1 has, each process has a user space and a kernel space, the user space is private. So, this is P1 space; this is P2 space, and the kernel space is shared right. So, this is so the kernel space is common. So, P1 and P2 both share the same kernel space.

The only thing that distinguishes two processes are the kstack. So, P1 kstack is different from P2's kstack right. And, we said that the state of the process within the kernel is encapsulated in its kstack right. So, the state of a process within the user space is encapsulated by whatever the contents of the address space are, but the state of the process inside the kernel. For example, what all the processes let us call where was where was it when it was context switched out; all this information is basically stored in the kernel stack.

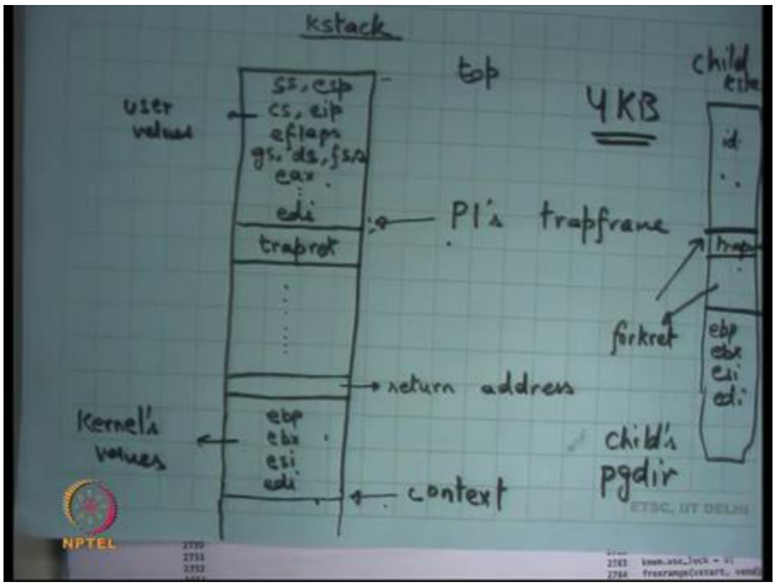
In general, I mean recall that we had made a statement that every user process is also kernel thread right. So, user process the state of the process involve also involves the contents of the address space, because each process has a different address space and different contents. The state of a thread typically is just represented by its stack right or can be encapsulated by pretty much its stack right. So, a thread basically shares the address space or threads multiple threads usually share the address space and each thread is really you know so what is we how do you distinguish one threads from another or what is the context of a thread, the context of a thread is really encapsulated in its stack right.

So, any scheduling algorithm among these threads is going to take one thread, if it switches out, it is going to just save all its information in the stack right. And it is gets switched in, then it is going to retrieve all that information from the stack. And we saw last time what kind of information can is saved and retrieved, the registers, the current value of the registers including the EIP, the program counter to which it should return immediately as soon as it gets context within.

Also, in the case of kstack, it also saves the information of the trap right. So, at the time that the trap occurred, what were the values of the user registers and all that right. So, kstack contains not just the information of what was happening in the kernel, but also what happened to the what was happening in the user at the time when the process entered the kernel right. So, it contains all this information.

So, when you switch to a new process, you can you know resume execution from where it was in the kernel space, and the stack also has in enough information, so that if you want to return to the user space, you will resume at exactly the same point from where you entered from user space to kernel space alright.

(Refer Slide Time: 03:21)



So, we look at the kstack in little more detail which we did last time, the kstack basically has you know let us say this is top of the kstack which means that is where you start from the, and on every trap and the trap could be system call, a trap could be an external interrupt, a trap could be an exception like divide by 0 or page fault right. You try to access a segmentation fault, you try to access an address that you are not supposed to access, in either case a trap frame gets pushed which contains all the users' value.

So, these are you know user values. So, if you want to return from here, you just going to pops the values, and you going to call irec which is going to pop the last five values and you going to resume back in your user mode seen as before. So, this, this pointer to the structure can also be called P1 strap frame right. So, this entire structure which contains the entire information about the user execution when it was interrupted or when it was you know when it entered the kernel, when it was trapped in the trap frame, and you can you know use this information to for example pass arguments from user to kernel we saw that.

And these registers can also be pointers, so you can actually even dereference these values because you are executing the same address space right. And you can also use this to pass to return values. So, you can change these values and you can say that you know I want to change e a x. So, you change e a x in the trap frame, and it is going to return the users going to see a new eax and that assume it is the return value of the system call for example right.

And the other thing that happens is immediately after this you know this is the return address. So, the stack has treated just like a normal function stack from then on. So, you just let say make a call to another function, but in doing so, you should ensure that the address the return address is pushed on the stack is the address of this function called trapret right. And trapret is going to execute these instructions which will you know pop all these registers and call iret right.

So, just to make sure that you know the, so from now on the stack will behave just like it is expected to behave in case of function calls right. So, you make a function call and, but before you do that make sure that the return address is trapret. Now, those functions are going to get executed, they could be a deep chain of function calls you know one after another in which case multiple return addresses will gets pushed. And when they return addresses will it popped and eventually when you reach here, then trapret is going to get popped and then it is going to pop off all these registers and then it is going to call iret, and going to get back to user right, right. ah

We also discussed that it is possible, so firstly the stack of a thread or the stack of a kstack of a process is finite right. The question is how big it should be right, clearly it cannot be infinite right. So, how does a programmer decide, how if you were a kernel programmer, how will you decide how big a case tag should be. Well, firstly, it should be at least as bigger the trap frame ok. Then it should now whatever function it is going to call what is the deepest function call that is going to have.

So, if you can analyze your code and say you know this is the maximum call chain or call depth that you can have, I will give you some indication of what the maximum size of the kstack should be. Moreover, a function should not have very large local variables. So, if you are allocating local variables on stack you know you should not be allocating like large arrays on the local kstack. So, all these things the kernel needs to be the developer needs to be careful about right.

So, few thumbs of rule, number 1 - you should not have very deep call stack, you should design your codes such that the call stack can never become so deep right; easy to do basically ensure that know the function call chain can never go beyond a certain depth.

Secondly, you never allocate large variables; you do never declare large variables at as local variables right. If you want large variables, what should you do?

Student: (Refer Time: 07:32).

Allocate on heap right, just malloc, just kalloc and allocate a page sized value and that is it right. So, do not allocate large variables on the stack ok, and that should pretty much satisfy what I said except there is one issue right. I said that a kernel while the thread was executing in the kernel mode and interrupt could come right. So, if the thread was executing in the kernel mode, and let us say with the stack pointer somewhere here, and another interrupt comes the timer interrupt comes, or you know a disk interrupt comes, or a network card interrupt comes, another trap frame gets pushed.

And while I am executing that handler, another interrupt comes, and another trap frame gets pushed. And theoretically I could very soon overflow my stack frame right, that is possible. So, what can be done to prevent this problem?

Student: Sir after three state frames you can call a (Refer Time: 08:30).

Ok, there is an answer that after three nested frames call a triple fault and fault I mean is that a valid solution, I mean it is possible that some external devices making lots of interrupts, do you really want to shut your system because some of your devices are misbehaving?

Student: Included, you know included.

Well, the answer is very simple. You basically ensure that some of your handlers are run with interrupts disabled alright. So, for example, any interrupt handler which hand which handles an external interrupt, for example, an external device interrupt like a timer or a disk or a network, these handlers will never run with interrupts enabled, they will always in run with interrupts disabled right.

So, notice that this, this kind of a situation where you know multiple trap frames are getting pushed on the stack can only happen if an external device is misbehaving right; it is giving me a lot of trap. So, if I ensure that any handler of an external device always runs with interrupts disabled while that a handler executing, I will not receive any interrupts right. The other thing I ensured is that my code is such that I will never you know I will never have more than let say n number of trap frames on the stack right for xv6 let say the maximum number of the n is 2 right.

So, one trap frame is the first trap, the first trap could have occurred either due to an external interrupt in which case they cannot be any other interrupt trap frame, because the entire interrupt external interrupt handler will run within traps disabled, or it can it can happen due to a system call in which case I do not need to really disable interrupts right. I want that my system call should be interruptible, and in which case an own, so now, now I want to make sure that my system call handler should never cause a trap itself.

For example, a system call handler should never cause a cause a cause a page fault or a segmentation fault, or my system call handler should never do a divide by 0 right that easy to ensure as a developer I can make sure the thing that is not in my control is somebody from outside taking an interrupt, so because of that I can have another trap frame. So, I can have at most two trap frames because the external device handler will run with interrupts disabled, I will not have more than two right.

So, that way you can limit the maximum number of trap frames you can have on this stack. You have also limited the maximum call chain. And from that you can estimate what is a maximum size of stack that you need. As on XV6, you just allocate one page for the kstack 4 KB alright, this is the question.

Student: so XV6 cannot support more than one external device because if there is if there is another external device that is also causing the interrupts, may be not able to support (Refer Time: 11:25).

Right. So, can you support more than one external device in this in this setting? Well, yes, you can, I mean what, what, so what happens if the interrupts are disabled, while the interrupts are disabled if another interrupt comes that interrupt gets ignored right, we have discussed this before. So, that interrupt just simply gets in ignored. What will happen if the interrupt gets ignored, does the does the device gets confused, because he is thinking he has given the interrupt, and the CPU has actually not received the interrupt.

Well, one thing you usually know the set setup is such that an interrupt needs to be acknowledged by the CPU that I have received your interrupt right. So, the hardware is you know is typically programmed in a way that if you have not received an acknowledgement, then you retry the interrupt let us say ok.

The other thing is to avoid too many retries the CPU itself has a buffer of you know one or two that interrupts. So, if you got an interrupt while the CPU was had disabled the interrupts, you just buffer the interrupt. And as soon as the interrupt flag gets enabled, you just push it to the CPU right, so that is just an optimization right. So, you will let us say you know you can buffer up to two interrupts let us say just so that the interrupts do not get lost ok.

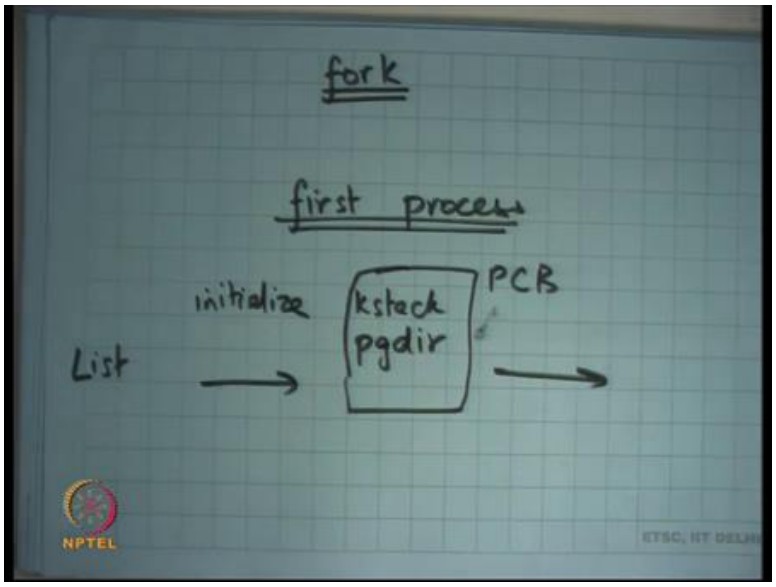
Another interesting piece, another interesting thing how does how does the computer keep time, how does it know how much time it has passed you know, how does it how does it keep track of you know how does it implement your clock?

Student: Sir clocks.

It just counts the number of timer interrupts. It has configured the timer interrupt hardware to say give me an interrupt every 10 milliseconds and just keeps counting the time number of timer interrupts that I have received and that basically tells in the time that is the only way it can so far we have seen that is the only way it for a for a CPU to keep track of the time. So, so with that we know that the kstack is a finite sized structure.

And when a process traps either due to a system call or a timer interrupt handler or anything else like an exception, it just creates this stack has a function chain. And let say at some point you want to switch it out in which case you are going to save the registers, and this is the case you are going to save the kernel registers value alright. And you are going to put this stack in memory, you know going to put this values in memory you are going to leave it as it is, and you want to switch to a new stack and that stack is now and you are going to unbind that stack just as you bound it, and you are going to and that basically implements your contact switch alright.

(Refer Slide Time: 14:27)



So, let us talk about how let us say fork is implemented alright. So, recall what is fork? Fork basically creates a new process which is a replica of the parent process. So, it creates the child process with the replica of the parent process, everything about it is identical in the user space except that the return value is different right. So, what does, what will I do, what do you think?

Let us say a parent process called fork. Fork will be treated just like as any of the system call. So, this stack frame is going to get pushed, some functions are going to get called. And in those functions, what is going to do is it is going to create another stack child kstack.

Where is it going to allocate the space to create the child kstack, from the kernels heap k alloc right. So, it calls kalloc to create a child case stack, and in it initializes the child kstack such that it has either identical trap frame right. So, the trap frame is identical. The contents of these two are identical id of this let us say. Except that.

Student: Kernel stack is.

The e x value is different right. So, that is that is how the child will have a different return value, and I will have a different return value that is all ok. The other thing I do is you know I also push trapret here, so that you know when it gets scheduled the it knows that it needs to pop of these values and then return to user mode. So, I will push trapret here. And then I will push some initialization function let us say you know some other return address on XV6, this is forkret, but let just say you know some function. And then I am going to save I do not need to save the entire call chain from the parent, I just save I just initialize that with the parents trap frame, the address of the function trapret and the registers which can be let us say zero initialized or something right. You do not care.

All you care is that you are going to return from here you are going to return from the trap, and you are going to start executing in user mode as the fresh process. Also, you are obviously, going to allocate a new pagedir and copy the contents of the kernels or the parents pagedir into the childs pagedir ok, so that is implementation of fork alright. Did I need to copy the entire call chain from the parent to the child?

No, that would have been wrong right, because the parent is executing a system call called fork. The child is not executing the system call called fork. The child just wants to return to the user mode exactly the same point that is all, it is not calling the system call called fork. So, I only need to copy this area; not, not anything below it. And I need to initialize something, so that it you know it starts from where it pointed start alright, so that is fork.

As I said you know all processes, so once a process has been created in future all new process is are created using fork except the first process right. So, the first process is special and that is something that is created by the kernel right. So, the first process will be created by the kernel. And what the first process is going to do is it is going to exact let us say some command which is going to and so you are going to see a shell on the console, and you can now type command and execute more forks and more exits right, that is how it is typically works.

There is one mid, one mid one init process and the kernel be it, be it xv6 or any main stream kernel, and that that just read some files and based on the files it just forks some processes in the beginning alright, but the first process is created by the kernel that is all. So, how is the first process created, very simple. In the fork, we just copied the stack from the parent to the child, if I want to create the first process, I just initialize the kstack alright and I initialize the fresh pagedir except there is a special kstack and special pagedir ok.

Student: Sir.

Yes.

Student: Sir like since the child and the parent now the now share the same page table. So, the pt and the (Refer Time: 19:06).

No, the child and the parent do not share the page table. The page table gets copied

Student: So, only the page directory is been copied.

No, the page by pagedir I am copying the pagedir I mean the entire two-level page table gets copied.

Student: Sir and the values of p t e underscore p are now set to 0 and that is because we have not set those values.

Are the values of p t e underscore p set to 0? No, I mean so basically you just so you copy the entire page table, and you also in fact copy the physical pages right. So, you so whatever is the size of the parent process, you do not just copy the page table, you also copy the physical pages right. So, you allocate new set of pages, you copy the pages from the parent, the contents of the pages from the parent to the child, you create a new page table to point to this new set of pages and that is it.

Student: So that earlier you said that there is demand page in.

Right, right. So, this is basically the naive way of doing things which is you copy the entire set of pages from the parent to the child, and then you initialize the page directory of the child. But we said the alloc this is expensive often not needed because the first thing a fork process may want to do is exec in which case you wasted all this work, in which case you can do demand paging right.

So, you know how will demand paging work? Well, I just you know I have the parent's page directory, and the page tables - two level page table. I copy the page table from the parent to the child. I mark all the pages in the parent's page table as read only; I mark all the pages in the child's page table as read only and I started running alright.

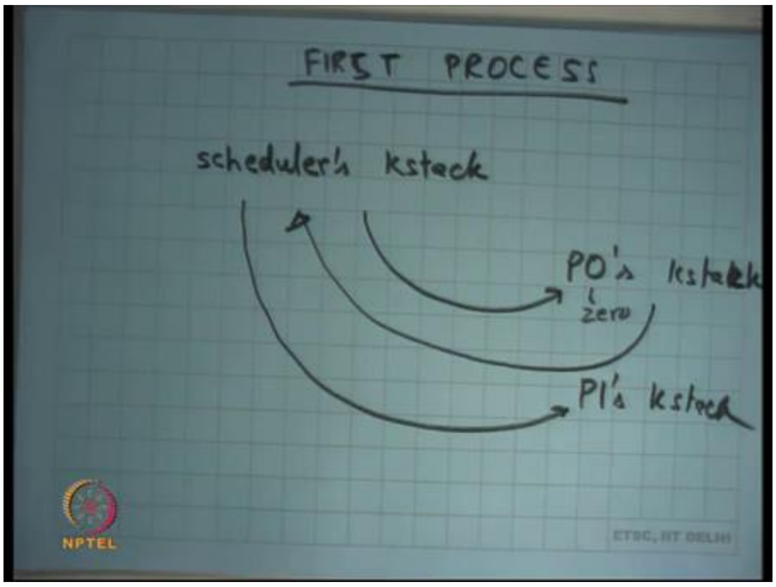
As and when the process is going to access a page that, with right intentions then you are going to get a page fault; the page fault can either occur in the parent or it can occur in the child. At the time of a page fault, you are going to look at which address it is that page faulted, and you are going to copy that particular page, and make two copies of it one for the child and one for the parent.

And you are going to remove the read only mapping to and make it read write, so that is an optimization. This called that that is a copy on right optimization and but let us keep things simple xv6 does not implement that. So, let just understand first how things work in the basic level and then we can definitely talk about optimizations from their own alright ok.

So, in both these cases, what I have done is I have initialized the kstack, and I have not done anything I just added it to my list of PCVs right. So, initialize, so creation of a process basically means initialize kstack and page dir. And, let us say put these in a structure called PCB right and put it to the list of PCBs. And just added to the list of PCBs and make it schedulable alright.

And then just call the scheduler, and the scheduler is going to pick one of these PCBs and just switch to it, because I have initialized the kstack in such a way that when you switch to it you are going to start running exactly as you wanted it to run that is fine ok, ok.

(Refer Slide Time: 22:33)



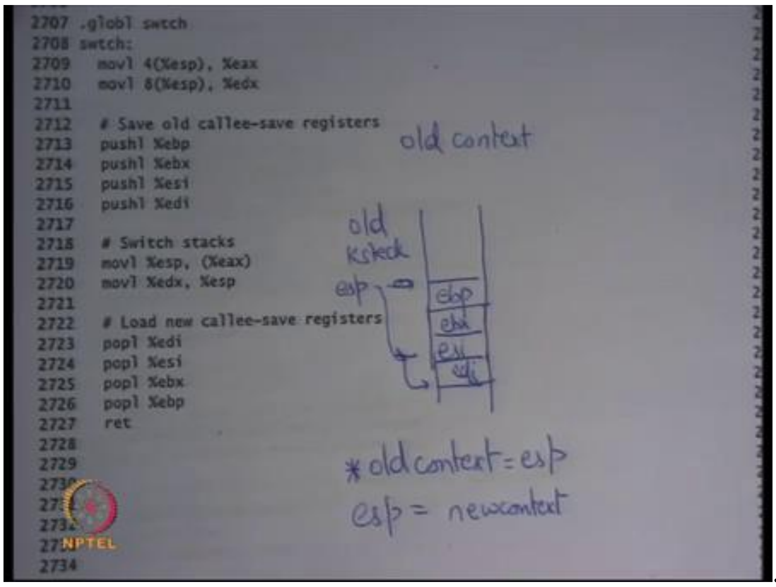
So, let us talk about creation of the first process. So, after booting up, the kernel has initialized itself to be running of a kstack, but this kstack is different from this kstack from the first process. So, let us call it the scheduler’s kstack right. So, the kernel when it initialized itself, it just initialized its kstack. And, it once again in the kstack was allocated of the heap, and that particular kstack does not belong to any process and let just call it the scheduler’s kstack right. Now, it is going to initialize you know P0’s, let say the init process of P0.

So, it is going to initialize P0’s kstack and its going to switch. So, this is 0 right, P0’s kstack, and it is going to switch from schedulers kstack to the P0’s kstack. The P0 will now get to run, and P0 may now let us say fork new processes. So, a new P1 gets created, and so new P1s kstack gets created right. And at some point, P0 is going to yield either voluntarily by calling the yields is call or involuntarily because of a timer interrupt. In either case, P0’s kstack is going to switch back to the scheduler’s kstack.

Scheduler is going to run by this is scheduler’s kstack is kstack that was running right in the beginning right. Scheduler is going to run on its own kstack, and it is going to pick up one process to run. And now it is going to switch to let us say P1’s kstack and this process just continues forever right. So, there is a scheduler’s kstack for a CPU, this which is which is first kstack that has started with.

It picks a process the logic of picking of a process executes on this kstack when you picked up a process you switch to that kstack that that process gets to run, then that process yields you switch back to scheduler’s kstack. The scheduler picks another process to run and so on right. So, this is a scheduler’s kstack. So, a process yields switches to scheduler then scheduler switches to yet another process and so, this seesaw keeps happening right ok.

(Refer Slide Time: 25:03)



```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

So, let us look at how the switch actually works. This is the first process that we are creating.

Student: PCO is the first process.

Yes.

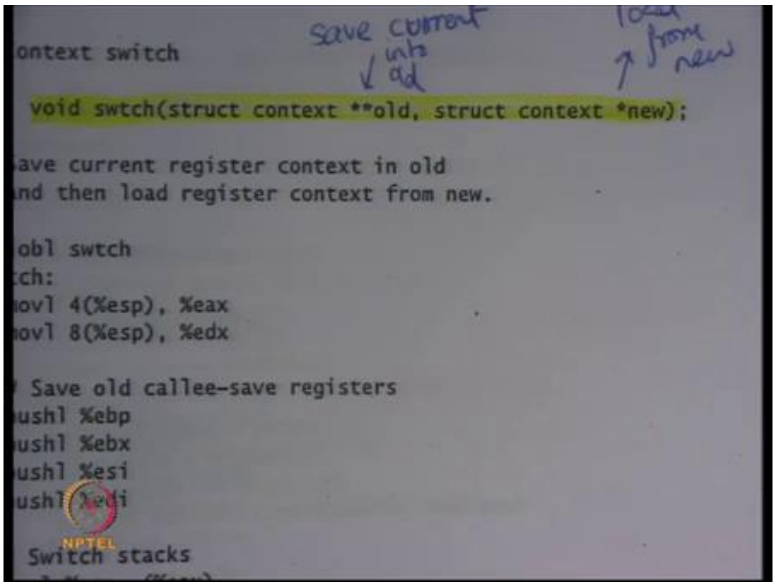
Student: Sir if scheduler implemented by hardware.

If the scheduler implemented by hardware, no, scheduler is completely software an operating system coded.

Student: Then it should scheduler be the first process or P0 be the first process.

The scheduler is not really a process right because it does not have a user spaced associated to it. A scheduler can be thought of as a kernel thread ok. So, every CPU just has this one special kstack that is all. And notice that in general there is a one-to-one correspondence between a thread and a stack right. So, because I am saying a scheduler has its own stack, you can think of a scheduler as a thread. A scheduler does not have a page dir, so scheduler is not a process ok.

(Refer Slide Time: 25:53)



```
ontext switch

 void swtch(struct context **old, struct context *new);

ave current register context in old
nd then load register context from new.

obl swtch
ch:
ovl 4(%esp), %eax
ovl 8(%esp), %edx

 Save old callee-save registers
ushl %ebp
ushl %ebx
ushl %esi
ushl %edi

 Switch stacks
```

So, here is the function which switches right. So, we said that the scheduler is going to switch to the new process. What does it mean to switch to the new process? You just call this function called switch right. Now, let us understand this function before we go forward. So, this is just function call switch without an array in this case, it is an assembly function. It takes two arguments there is a structure called context which basically holds the saved register the values of the kernel right.

Recall that we said that some values that the kernel has and needs to be saved. So, it takes a pointer to the context. So, there is a structure called context, this, this is an old context, and this is a new context. And what the switch function is going to do it switch from the current context to the new context and save the values of the current context into old alright.

So, the semantics are saved current into old right, and load from new ok, so that is what this function is going to do. It is going to save the current context which is the value of the current registers into old and load the values of registers from the new context the then it is done basically alright.

So, where is this context one gets stored? So, what is this context? If I look at this figure here in this kstack alright, this is my kstack. Then the context will be this pointer here ok. So, it is going to save values at the bottom of this stack that is with a that is where the old context is. So, if you are switching from here to here, then this will be the old context.

And you are going to save the current values here, and you are going to load the new context from here right, so that is what this, this going to do going to, save the current context into the old context, and the old context will live on the kstack of the old process or old thread. And similarly, you are going to load values from the new context.

So, well, what does it do it, basically just takes its first argument. The first argument is at offset 3 from the current esp, it is a function. So, you basically using the function calling conventions gcc calling conventions which are in the first argument is that offset 4 from esp put into eax that is your old pointer. This takes the second argument 8 offset put into the edx that this is this is a new pointer. You push values on the current stack whatever the values of the registers are on the current stack, whatever current values are on the stack. And you move this stack into the old context. Recall the eax was the old context. So, you move values into.

So, basically what is happening is there is a pointer called old context right, and there is let us say old kstack. So, currently esp will be pointing in the old kstack somewhere, you push some values you push the values of the current registers, let us say ebp, ebx, esi, edi. esp now starts pointing here right, and you basically save the value of the esp in old context. You save old context is equal to esp, star old context is equal to esp. So, you save the value of the current esp into old context alright. And then you say esp is equal to new context alright, so that switching taking place. You saved the current value of esp into a pointer called old context, and you loaded esp from the new context right.

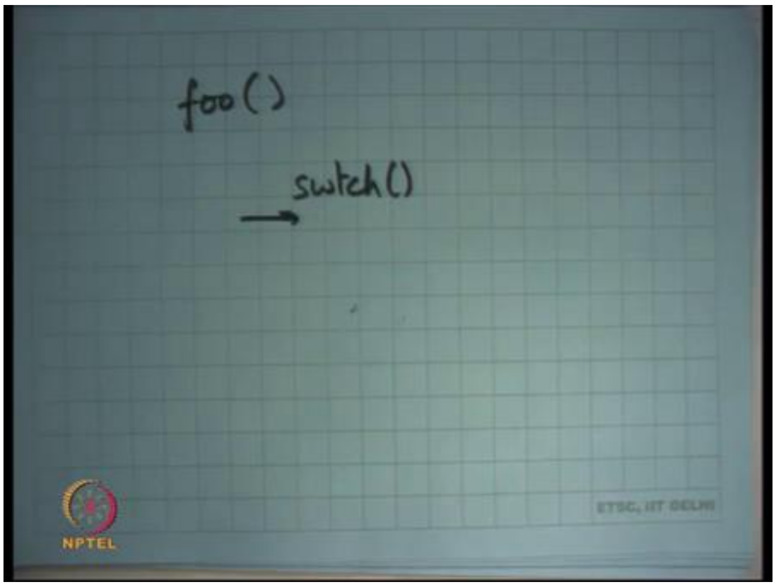
And then whatever you did here you do the opposite of here which is pop all the registers. So, because the old context, the new context must be assumed to be in the same structure as your current also. Assuming that that process was also switched out in the same way which means it must have pushed all these registers. So, the moment I switched to the new context, I can now start popping those registers, and I can call the return instruction which is going to go to the caller of switch and continue like that right.

So, that is where you switch the stack and basically all stacks, so all saved stacks in the kernel will always be in the state with the last four values in the stack will be these saved values ebp, ebx, esi, and edi always right. And so whenever you switch to a new to a stack, then the first thing you will do is pop off all these values, and the fifth value will be the return address, so return will return to that address right.

So, basically when I was talking about switching kstacks, let us say this is my kstack, I have now I have a new in variant that context will be pointing to a location in this my kstack, such that the first four values will be these registers, and the fifth value will be return address. And so, whenever I switch to it, I will always switch to it in a switch function. And, when and what I am going to do is I am going to pop off these values and then call return, and I am back in business I am back in action just like I left. So, irrespective of where I am, I call switch, I get switched out. At will some later point, I get switched in and I start resume execution resuming from exactly where I left right.

So, whoever called switch is going to now it starts executing from the next instruction after calling switch right just like before, so that is a context switch, just with the stack right. So, one thread enters a switch, and another thread leaves a switch right. So, it is a special function in that sense. Typically, you enter you know you enter let us say you let us say foo called switch.

(Refer Slide Time: 33:05)



Let us say this is foo, and it calls switch. And let us say there was another function, so it is probably never going to return here right. It is going to return in some other it is going to start executing some other thread and then that thread is going to call switch at some later time, and the that time I am going to start resume execution from here right. So, I do not immediately return from switch. I return from switch much later on another context switch right that is a difference basically alright. So, the switch function has not changed the eip value. How is the eip value getting changed?

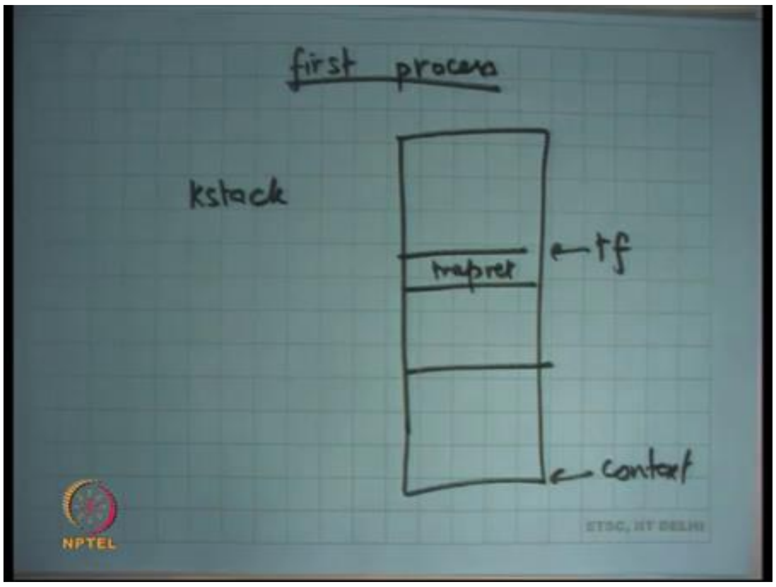
Student: The switch.

By the return instruction.

Student: Return instruction.

Right. So, the return instruction is the one that change changes the eip value. So, the moment you switch this tag you also change the return address alright. So, that is the trick going on here. You change did not change the pointer you did not change eip explicitly, you just change the stack and the return address got changed right. So, this is this is very interesting you know this is by this is the heart of an operating system which is the context switching code and the quite tricky in that sense alright ok.

(Refer Slide Time: 34:33)



So, what does this tell us, it tells us that if I want to create the first process, I should initialize the kstack such that it has the trap frame, it is called t f at the top, it has trapret here; and at the bottom it has the context right. So, that is what I am going to do. I am going to initialize a kstack which looks just like this; I am just added to the list of PCBs and called the scheduler. And the scheduler is going to call switch, and because the kstack is well formed just going to start returning from here and start executing the first process ok.

So, the invariant I am maintaining is that all the kstack's that has saved that are not currently running are always in this sort of format where the last few bytes, the context is pointing to a location which is basically pointing to the size of the current stack. And the first few bytes after context are basically the saved registers and the return address ok. Question, why am I only saving these four registers, why not eax, ebx, what happen to those registers?

Student: already saved in the trapret (Refer Time: 35:50) this switch was (Refer Time: 35:52).

There is a there is an answer that they were saved in the trap frame is that right? No, I mean trap frame saved the users eax, ebx, now I am executing the kernel, so kernel has some e a x e b x also. So, I they may have changed. So, why am I not saving eax, ebx, any?

Student: (Refer Time: 36:08) caller, caller, caller.

Yeah, it is a caller saved register right. I am assuming that switch is a function call and I am assuming that these the function calling conventions have been weighed, so all those registers must have been saved by the caller. So, only need to save the callee save registers ok. I mean I could have saved all registers that would have been just extra work unnecessarily right, because column must have saved then for me anyways. So, I just need to save the callee save register ok, very good ok. Student: Sir.

Yes.

Student: Sir, what who is the callee here.

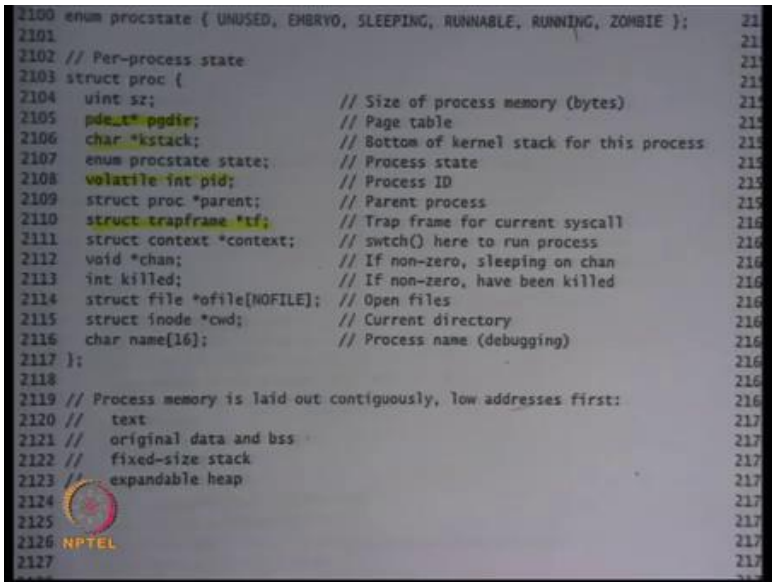
Callee is switch; callee is the switch function.

Student: Fine. So, whenever I have formed the (Refer Time: 36:58) all the switch function and then the function the eax and all gets stored.

Yes. So, ok, the question is who is the callee and who is the caller? When, whoever, whoever was, whoever called a switch function is the caller, and because he is falling the calling conventions, he must have saved the caller saved registers right. And he must have ensured that you know when it will come back from the switch function, he will reload the caller saved registers back.

So, I do not, so as the function I only need to worry about, but the callee saved registers is I am a function right. And I am assuming that the function calling conventions have been obeyed alright. So, then let us look at then let us look at sheet 20 and let us look at the structure of the pcb alright.

(Refer Slide Time: 37:47)



```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
```

So, this struct proc basically is the structure of a pcb process control block inside Xv6. And here all the fields of the process control block. So, let me bring your attention to some of the fields that you already understand. So, here is a pointer to the page directory right. This pointer value will always be a low address or high address I mean will it be an address above kern base?

Student: High address.

It will always be a high address right, it will always be a kern base and above address alright, because it is something that you allocated and now you are going to load the value of this pointer after converting it to its physical address by subtracting kern base and loaded into c r 3 whenever you want context switch to it right, so that is what page dir does right. And pages for the page dir and the all the second level page tables are allocated of the kernel heap right.

Kstack, we understand this. Once again, allocated on the kernel heap and the maximum size of kstack in Xv6 is one page. So, we understand this also, good alright ok. Then there is this integer called pid which basically says what the process id. So, every process should also have an id right. And so, and there is a system call called get pid which allows the process to know its current process id, so you need to store that alright.

Then you see that there is this pointer called struct, trapframe star t f. What is it? It is a pointer into the kstack at the location where the trapframe is stored. Why do you need it? If any of the downstream functions wants to access the values of user registers, you can just do t f pointer dot you know value, and so it is very nice easy to do that ok. Then there is a pointer to the context.

Student: Sir.

Yes.

Student: So, t f stores the valid value only if we are under a system caller only if we have only if we have executing in the kernel space right now I will.

So, question is t f stores the valid value only if we were executing in the kernel space right now ok, what does it mean? So, does t f always store a valid value? Well, if the process is not currently running, then the t f must hold the valid value right. Because if the process is not currently running, it must have it switched out in the kernel space right; if the process is currently running, then yes t f is not valid.

It is not valid if it is if the process is currently running in the user space, then t f is not valid; if the process is currently running in the kernel space, then t f is still valid right. But for any process that is not running, so any saved process or any you know suspended process t f will always be valid right.

Because the suspended process, in fact, both t f and context will be valid right because as we said any suspended process processes kstack should look exactly like what we drew earlier. And, so it should have a context pointer, and it should have a t f star frame pointer. And, so when you want to switch to that suspended process, you are just going to load the stack from that process is context.

So, you are going to look at that process is context, and whatever the value pointer is there you are going to loaded into esp. So, that is what context is storing. You switch to this context ok, ok. So, this is also very important the context field is important, because when you context switch you switch to this context which means you load the esp with this value.

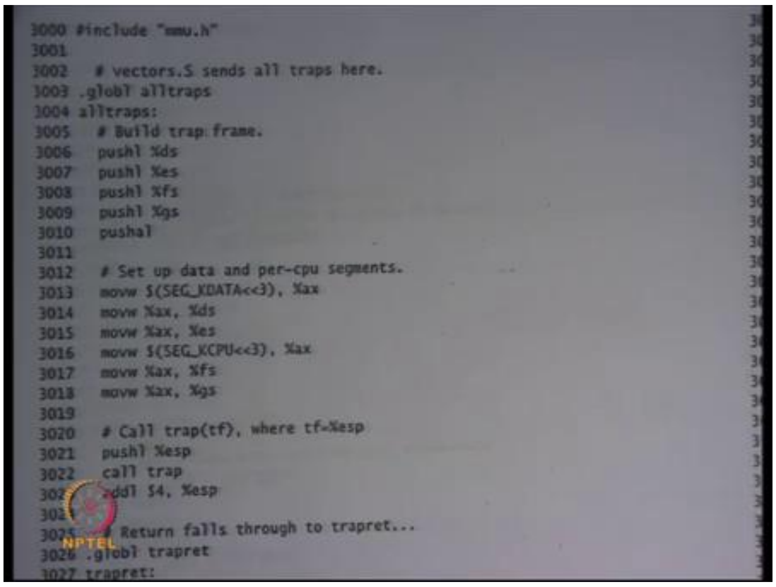
So, the scheduler has figured out that this is the process that I want to run next it just looks at the context field and call switch with this as the second argument and so switch loads that value into esp good, alright, and alright. So, those are the relevant field so far.

But let us just look at the other fields let us say state. So, a process also has a state. It basically says whether it is used or unused, so this is just to say that a process whether it exists or not alright. So, if you not implementing a process, the list of pcb is as a list, but you are implementing as it an array, then you can just have this bit saying this, this pcb is not used which means that is not even allocated.

Embryo, ok, what embryo means we are going to discuss later by its basically means the process has not even get bond yet right. So, its juts actually just called fork recently, and the process is just sort of getting initialized. And so only when I am going to add it to the pcb list, am I going to change its name from embryo to embryo to runnable alright. So, runnable says here the process that can be run right. So, any runnable process must have its case tag initialize exactly as the way we discussed right, so that is the invariant. Any runnable process any process whose state is runnable, its kstack should have been initialized exactly has we discussed basically.

A process could be running its currently running right. So, you know these pointers may not make any sense that it is running in the user mode, then kstack is not useful alright; k stack just points to the bottom of the stack actually, but let us say you know the t f and the context fields are not useful alright. And, then there are other fields like sleeping and zombie which we understand but let us let us ignore them for now because we are discussing other things right ok. So, let us also look at what happens on a trap sheet 30 ok.

(Refer Slide Time: 44:07)



```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
```

So, on a trap the idt points to handler right. And we said the first thing the handler does it save all the registers alright. So, the way xv6 is organized, it is that it has pointed all the idt handlers you know it has some initialization code for every handler, but eventually they all jump to this function this code pointer called trap, we just call jump here not call, but jump alright.

So, this is just jump here. And here you see all the code for saving all the registers. So, you save all this segment registers. And this instruction push al basically saves all the general purpose registers like ex, ecx, so all the eight registers that we have discussed. So, that is push all that is what push a means. And then it loads the segment registers with the kernel segment right because the difference between SEG_KDATA and SEG_UDATA is only that.

Student: (Refer Time: 45:01).

Different privileges, that is alright, base and offset are identical, base and limit are identical. So, we just load it with kernel segment. You have saved the user segment; you load the new kernel segment. You do it for all the registers segments registers ok. What seg k CPU is let us ignore it for a moment let just say it is you know there is something that the kernel is doing for a k data and k CPU. It pushes the current esp also. So, push a does not push the right esp. So, you will also push the esp. So, one register got one register is not correctly pushed by push all which is the esp register itself, because you can imagine esp is changing right as you are pushing throughout.

So, so will have to specifically use as instruction for call push esp, and then you call a function in c which is called trap right. Trap will basically figure out what was the why I why I trapped and all that, for example, let me look at the trapret to figure out why I trapped. So, what is happened is this function has created a has initialized the trap frame right. And then it has called trap, trap will return at some point and you are going to get back to trapret right. So, in this case, the code as been nicely organized such that you call a function, and when it returns it actually returns back to trapret, and trapret is just going to pop all these registers and call iret.

Student: Sir, when it will return it will return to trapret or it will return to line three 302.

Yes, it will return to line 3023, and 3023 is just ignoring the esp value that is alright. And then it calls to all trapret right. So, yes, I mean trap will return to line 3023, 3023 is not doing much, it just ignoring one value on the stack it just popping of 1s value on the stack ok. And then it calls pop a l and all the other things.

Student: Sir aren't traps the low return functions like they never return the (Refer Time: 47:02).

Ant traps no return functions, well I mean this trap function is not a no return function, it returns.

Student: Sir, but if it returns why do we need a specific (Refer Time: 47:18) for trapret as a nobody is going to call trapret, it will automatically come here and follow the.

Right, right, so that is what is happening. So, in this case in this special case or in this general case I should say, whenever a trap happens through the regular way, then trapret automatically gets pushed as a return address or you know trapret minus you know whatever the 3023 gets pushed as a return address and so everything works normally.

But when I am going to create a new process, what I am going to do the new process never call the trap right. So, I am going to simulate as though that, so, I am going to push a trapret manually in that in general the trapret gets automatically pushed, because you did a call and you made a call. So, then the next instructions the address gets pushed on the stack. But when I am going to create the first process or when I create a fourth process a child process, in all these cases, the child process did not make a trap right. So, I need to initialize it with trapret. So, either way they should be a trapret on the stack frame just below the trap frame right.

So, the stack is basically the trap frame and trapret. In case of a normal trap, it automatically happens paired by the call instruction; in case of the first process and a fourth process, we need to do it manually that is all.

Student: Sir when we do a fork.

Yeah, I mean when we do a fork process and or a or a new process, then we do not push esp at the end. So, they we do not need the statement let us say yeah valid point.

Student: So, why are we saving all the registers like why are we not calling the caller, callee conventions here?

Ok, great, great question. So, why are we not saving, why are we saving all the registers, could I have used the caller callee conventions in this case just like switch? The switch was a function; this is a trap. A trap follows no conventions right. I need to save all the user registers. A user did not, users there is no we never studied a, we never discussed any convention that before making a system call, the user will save some registers. If there was such a convention, then yes, I would have saved less registers.

But, even then you would have had to have a different handler for the system call and what happens if there was an external interrupt you know if there was an external interrupt the user was completely not expecting a external interrupt. So, he may have all the registers valid and so I need to save all the registers. So, there, firstly, there is no convention on sys calls, and then we need to worry about exceptions and interrupts. So, we need to save all the registers alright.

So, let us stop here, and we are going to discuss this more the next time.