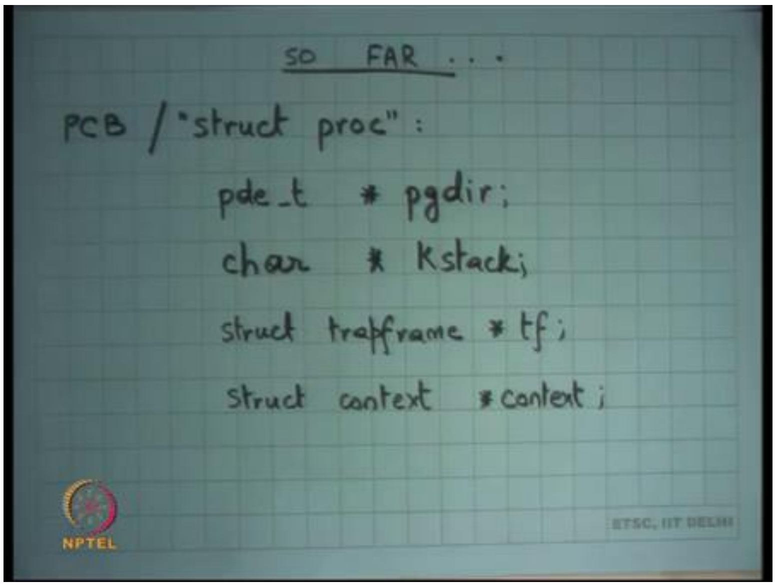
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 19
Creating the first process

Welcome to Operating Systems, lecture 19.

(Refer Slide Time: 00:31)



SO FAR . . .

PCB / "struct proc" :

```
pde_t * pgdir;
char * kstack;
struct trapframe * tf;
struct context * context;
```

So far, we were looking at how the operating system represents and manages process. So, we said that an operating system has a structure called PCB in the process control block for every process. On (Refer Time: 00:41) this is the struct proc that we are looking at last time and some of the important fields we looked at were you know each process has a pointer which is of type page directory entry.

So, this is a pointer which will be a page sized which will point to a page sized allocation which will be the page directory right. And, that page directory will have more pointers to the second level page table and those page tables will have pointers to the actual pages and so, this entire structure is private to every process which means every process has a private page directory, private second level page tables and private pages right; assume assuming we are not doing optimization like copy on write etcetera.

I mean conceptually speaking each process has a private address space. So, it has a private hierarchy of the page table and the physical pages themselves. Each process has the private stack kstack right and once again kstack is allocated from the kernel heap and it is also a page sized entity in the xv6 kernel. Just to and we last time discussed why a page sized kstack is enough. We said the programmer can be careful such that stack never goes beyond one page right.

And, the way that it can do that is basically ensure that the call depth is bounded, that the mount the size of local variables is bounded, that asynchronous interrupts cannot be received while an asynchronous interrupt handler is running right. So, you cannot just have asynchronous interrupts causing trap frames to get pushed right. So, with these three things basically ensure that kstack remains bounded and once again kstack is private per process.

And, so just to give you some fact point a point of data Linux kernel for example, uses a kstack of 8 kilobytes right. So, not really big right xv6 uses 4 kilobytes, Linux use 8 kilobytes and this not of stack is usually sufficient for whatever the kernel wants to do alright. This is a yes, there is a question.

Student: Sir, however, supposed you write a recursive program, then it is like if the programmer if let us say the designer is just writing a recursive program then it depends up on the user whether or not the what this stack length would be right. So, if let say I am writing a simple factorial program and I give him an argument of let say hundred. So, if my stack depth is not that much then would that lead to a stack overflow?

Yes. So, let say you know a programmer writes a recursive program let say he just simply writes a recursive factorial program and your stack depth has limited by whatever you have said here that is a page size, then will your stack overflow? Yes, that it will overflow. So, what does that mean? The kernel developer should never write a factorial program inside the kernel right. There should not be a function which is you know recursively going to unbounded depth depending on the user input it should not be possible right.

A user program also does not have an infinite stack right. So, even if you write a user program has a larger stack than the kernel the kstack, but it is not infinite either right. So, even if you. So, there is actually a limit to how much recursion you can do even in the user space, but it is much larger than what you can do in the kernel.

Student: Sir, we said that we disable interrupts value we are handling the external interrupts. So, while we are handling the external interrupts can a process switch happen because of timer interrupt?

Well, I said that we disable interrupts while we are executing an interrupt handler and when the interrupt handler is running can you actually switch processes right or switch the kernel stack and then you can move the, yes, you can right, but it will this switch will not re-enable interrupts. Interrupts will get re-enabled only when you go back to user mode.

Student: But then switch happens on the timer interrupt.

Sure. So, what will happen is let us say process P1 is running and an interrupt occurred process P1 starts running in kernel mode on the kernel stack with interrupts disabled. The kernel stacks get switched. The new kernel stack P2's kstack is still is to P2 starts running in its kernel mode on P2's kstack, but still these interrupts will be disabled.

Student: Sir, is it not the timer interrupt disabled?

It will be disabled for the duration of the switch right and then as soon as you go back to the user mode you are back again, right. So, how does a timer how do the interrupts get enabled when you go back to the user mode because when you actually do return from trap so, you pop off all the registers and then you execute the IRET instruction.

And, recall the IRET instruction pops off EFLAGS right and so, the flag in EFLAG which indicates that the interrupts are disabled or enabled gets reinitialized based on the user value. And, so, now the this when you are in user mode you are executing with interrupts whatever the you know in general you will be executing in user mode with the interrupts enabled right.

Of course, now the user if it makes a system call then now the system call need not disable interrupts, but if another time interrupt occurs while you are executing in user mode then again you will execute with timer interrupts disabled right. Interesting so, another question I have for you can use should a user mode program be allowed to disable interrupts?

No, because if a user mode program is allowed to disable interrupts, he can just take control of the system right. So, how does a how does architecture ensure or how does the OS developer ensure that a user mode program cannot disable interrupts?

Student: Sir, you cannot modify the (Refer Slide Time: 06:28).

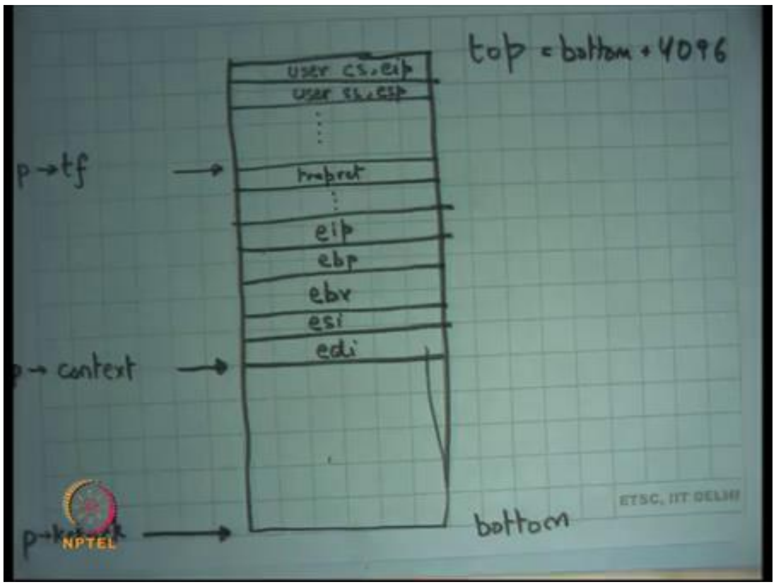
So, the CLI instruction that is the only way to modify the bit of the EFLAGS is the privilege instruction. And, so if the user ever executes that instruction what will happen? An exception will get thrown and the kernels exception handler will get called and the external exception handler may just kill the process for what you kept right. There is another question? Alright ok. So, that is kstack.

And, now trap frame. So, we said that each PCB on xv6 also has the pointer to a trapframe and this trapframe is basically nothing, but it contains all the registers and in user mode just at the time of entry to the kernel right and this is a nice thing to have because later if you want to refer to these registers for either arguments or anything else or return value then you can do that using the trapframe. And, trapframe is always pointing within this stack within the kstack right. In fact, we are going to see exactly where it will point all the time.

A trapframe for a same process is always valid; a trapframe for a runnable process is always valid a saved runnable process is always valid; a the trapframe for a running process a process that running in user mode specially is not valid because it will get overwritten as soon as you come from user mode to kernel mode so, whatever the contents completely immaterial alright.

And, then we said that there is another pointer called context and this context is basically saving the state of the kernel thread of that process. So, when you came into the kernel mode you executed some functions and then at some point you call the switch function which we saw last time. And, the switch function is going to create this structure called context where it is going to save all the quali saved registers and the return address that is the eip on this stack point context to that location and then switch to another processes context right.

(Refer Slide Time: 08:23)



So, if I look at if I look at the stack of any process case stack. So, let us say this is the p kstack I am calling this is the bottom this space from context to kstack is not used right and this is the top of the kstack. So, this is you know what top is equal to bottom plus 4096, 4kb right. So, we know that it is you know it is a fixed size stack.

So, you k alloced a stack so, that is what you got to the pointer to kstack then you said you know plus 4096 that is my top that is what you are going to feed into the TSS and so on every trap all these values are going to get pushed first by hardware and then by your interrupt handler instructions and that is way you are going to set up your trap frame pointer right.

So, this is the trap frame pointer and you can look at all these fields from the trapframe. Infact you did not really need the trapframe pointer because you already know if you know kstack you can calculate tf by simply saying kstack plus 4096 minus whatever is your size of trapframe right. It is always it always at the constant offset from kstack, but just for convenience let us just have a separate pointer in the PCB not a big deal.

Also, we say that and also these values get pushed by the interrupt handler and the interrupt handler ensures that the next sort of return address should be this address of this function call trapret right. And, we said that one way to do that is just make sure that arrange it on the function and then make a call to wherever you want to do. So, the return address automatically ones call it trapret and then you will you know call some function which will have their own local variables and return address etcetera.

And, finally, somewhere you here you will call switch right. So, as you as soon as you say call switch the return address of the function which call switch gets pushed as the eip right and then the last next four callee save registers get saved by the switch function itself and that is it right that is where you have switched at this point of the stack you actually perform the switch right.

So, for any runnable process this will be basically structure; context will be pointing here. Everything here is completely unused and everything here is meaningful and when I am going to switch, I am going to switch to the other processes context and I am going to pop up these values to refresh my kernel to reload my kernel state and the kernel is going to do something and then eventually it is going to pop these value to reload the user state right.

(Refer Slide Time: 10:56)

THE FIRST PROCESS

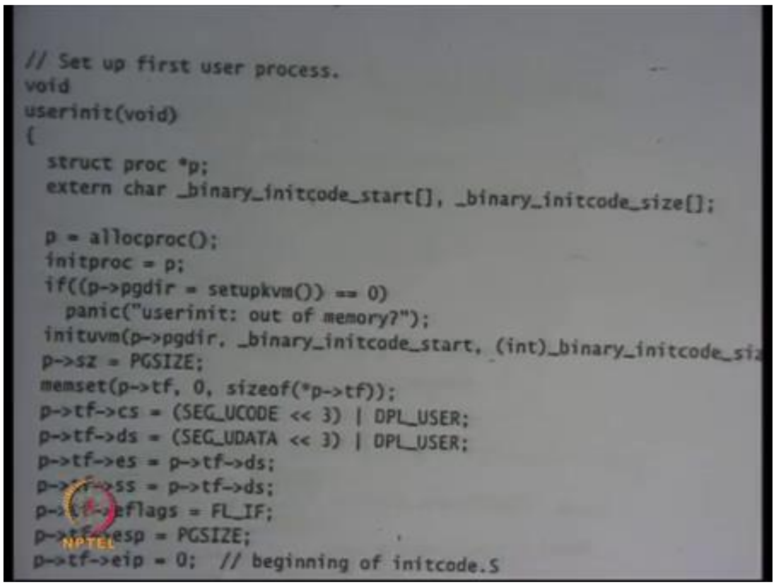
userinit

So, today I am going to talk about I am we going to look at how xv6 creates the first process right. So, basically what is going to do is there it is going to allocate a page directory and it is going to initialize the address space to point to some code that the kernel wants to execute first in the user mode, that is one thing it will do. Then it will allocate kstack and it will organize the kstack as though the process was just switched out right.

And, then it will add the, So, it and it will initialize the PCB to point to all these things kstack and page directory and other things and it will just add the PCB to the list of schedulable processes; in other words it will just mark the PCB runnable right and then it will call the scheduler. And the scheduler will just it will go through the list, pick up the any runnable process and in this case, it is the only runnable process because of the first process that you are creating.

And you just pick it up because you had set up the stack in exactly the way any other process would have been had it been switched out it just starts running in the normal way right ok. So, let us look at that. So, we are going to look at this function call user init on sheet 22.

(Refer Slide Time: 12:15)

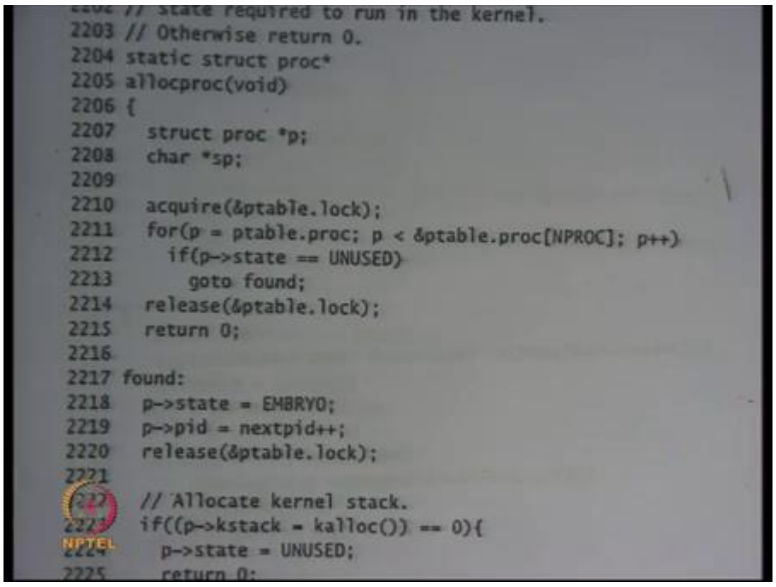


```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_si
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S
```

So, this function is called from main after other things have been initialized and user init is going to set up the first user process right. And, what does it do? It makes a call to this function called alloproc. What is alloproc going to do? It is going to allocate a PCB.

It is a going to allocate a PCB and it is going to let us see what exactly alloproc does it is going to allocate a PCB and it is going to allocate the kstack of that PCB and it is going to initialize the kstack with a trapframe etcetera alright. Let us look at alloproc first.

(Refer Slide Time: 12:51)



```
2202 // state required to run in the kernel.
2203 // Otherwise return 0.
2204 static struct proc*
2205 allocproc(void)
2206 {
2207   struct proc *p;
2208   char *sp;
2209
2210   acquire(&ptable.lock);
2211   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2212     if(p->state == UNUSED)
2213       goto found;
2214   release(&ptable.lock);
2215   return 0;
2216
2217 found:
2218   p->state = EMBRYO;
2219   p->pid = nextpid++;
2220   release(&ptable.lock);
2221
2222   // Allocate kernel stack.
2223   if((p->kstack = kalloc()) == 0){
2224     p->state = UNUSED;
2225     return 0;
```

So, on the same page there is allocproc line 2205. All it does is it assumes that there is a global variable called ptable – this table of all the processes. It iterates over the table; so, I have been saying that it is a list of PCBs, but xv6 implements it as an array of PCBs right. So, it just says that there is a limit to the maximum number of processes you can have in the system that is NPROC. So, it is just going to go over this array of processes.

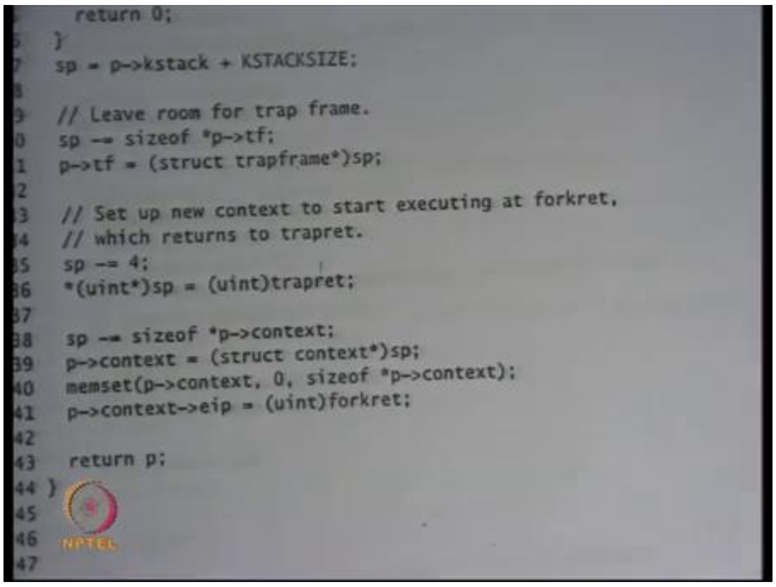
And, if it finds that one of those processes has a state unused then it is going to do other things. If it could not find anything that was unused, then it just says that I could not find anything. And, so what should happen is basically if the first process was not been able to create get created then you should just return an error that you could not boot properly basically; if you came here because of fork, then you would just return a minus 1 value to the fork right ok.

So, at this point you have allocated you found the process that you found space for the process you found space for the PCB you initialized state to EMBRYO. So, you initialize it. So, from unused you initialized to EMBRYO. The reason you initialized to EMBRYO is because just in because xv6 is the multi processes system.

So, once you have taken up you have pinned one particular process block and said that I want to use it you want to basically mark it such that nobody else starts using it after that right. So, exactly how this works etcetera will be discussion of concurrency and locking which we are going to do next but let just notice this at least right.

Then we assign pid to this process. So, this is global variable called next pid which is let us say initialized to 1 or 2 or whatever and each time you create a process you just increment pid and just put it in this variable right and then you release a lock we going to talk about locking later, but let just not worry about it now alright.

(Refer Slide Time: 14:50)



```
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  return p;
}
```

Then, we allocate stack kstack right. Once again if you could not allocate a stack you return an error and you reset the state of the process to be unused. So, you know you did not actually do anything and fork field and then you basically initialize the stack pointer. This is a local variable sp to kstack plus KSTACKSIZE and then in our case kstack size is 4096 alright.

And, then from there you make space for the trapframe. So, size of star p pointer t of f is basically saying size of struct type trapframe. So, basically say you know that is the location where you are going to point tf to right. As I told you that you know kstack and tf are always at constant offset you do not really need to store it, but it is more convenient to store it alright.

And, then and then what is he doing is he is basically saying let us decrement s you let us push the address of this function called trapret right. So, we saw this function called trapret, this just pops all the registers and then calls IRET. So, just pushes the address of the trapret function into the stack that is what is it is doing.

So, sp minus is equal to 4 and star is p is equal to trapret that just pushing trapret into the stack and finally, it is making space for a context structure right. So, it just makes space for a context structure. So, size of star p pointer context. So, let us I am going to show you this context structure context structure is nothing, but those five registers four callee saved registers and one instruction pointer eip right.

So, all those five registers space is created for and you make context point to that location. Is context also always at a constant offset from kstack? In this case it is right because you can say that context is always equal to kstack plus kstack kstack size minus size of star tf minus 4 and that is minus size of star context and that is why context is point, but in general.

Student: Always.

Is context always at a constant offset from kstack? No, right because you may have some call chain between trapframe and actual call to switch. But, in general actually it is the trapframe also at always at constant offset from k kstack? Not necessary because as the first trapframe will always be at a constant offset from the kstack, but if there was another trap while you are executing in the kernel then that trapframe can be anywhere in the kstack right.

So, actually you know trapframe is not the tf variable is not redundant. It is basically it is telling you the location of the last trapframe alright. So, it is not necessarily at a constant offset from kstack, it is actually could be somewhere in middle alright.

So, this is context and then you. So, here what you are doing is he is zeroing out the context structure except that the eip of the context is set to the address of forkret right ok. I am going to discuss this very soon but look at this first I did not really initialize the trapframe right. I did not I only allocated space for the trapframe. I did not say at the trapframe register ei should be this value or cs should be this value or ss should be this value they just you know they are just random values here I just allocated space.

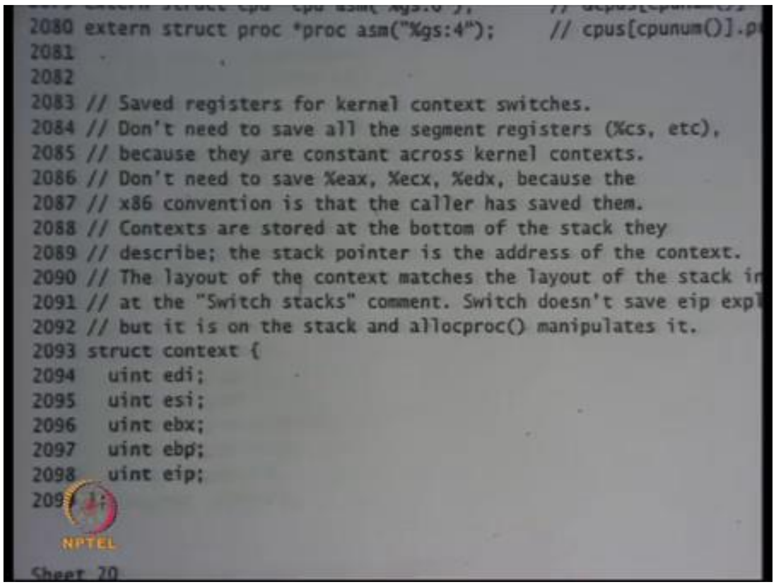
So, this function call allocproc is not initializing the trapframe, it just allocating space for the trapframe and whoever is the caller is supposed to fill values in the trapframe. So, in the case of user init, you are going to manually fill some values. In the case of fork, what is going to happen?

Student: The parent just would.

The parent processes trapframe contents are going to get copied into the trapframe right. So, the allocproc is not doing any initialization of the trapframe it just allocating space and the caller depending on whether it was a fork or whether it was a user init it is going to initialize it differently alright.

So, context also has not been initialized except there is one value in context that was initialized that is the eip all other registers are being initialized to 0 which are which is which has no meaning really right.

(Refer Slide Time: 19:09)



```
extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].p


// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack i
// at the "Switch stacks" comment. Switch doesn't save eip expl
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
```

Sheet 20

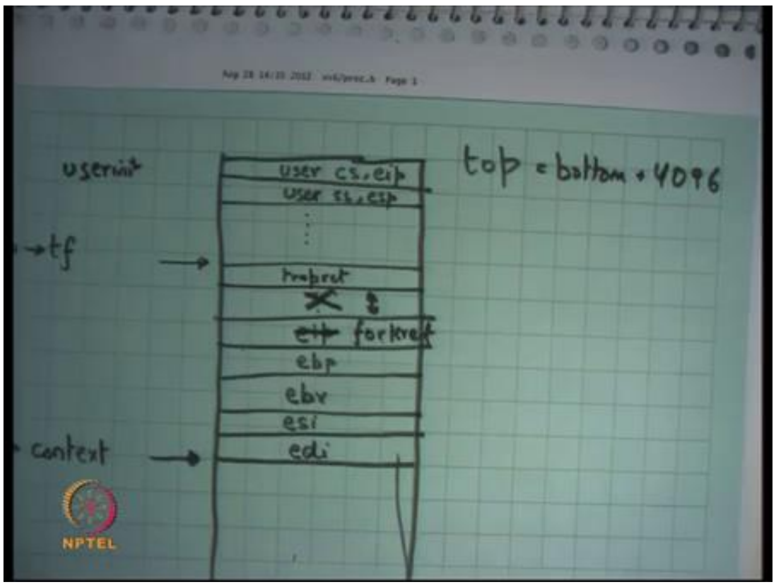
Let us look at their context once again refer sheet 20. This is the declaration for struct context and struct context has this five fields which basically look they are integer fields and they have been named on the registers that register values or register names edi, esi, ebx, ebp and eip alright.

Notice that this these fields occur in the inverse order in which they were pushed in the switch function alright. So, for example, you have pushed ebp first so, that comes last. In fact, you push eip first because as soon as you call instruction the return address got pushed first. So, eip is last, then you push edp that second last ebx, esi and the last register you pushed was edi. So, that is a first alright.

So, it is you know if you look at this stack you pushed in this order and you pointed context here and so, context basically you if I was to if you I was to get the edi value by the first field of context. And I would have to get the eip value that is the last field of context right and so, what I have done in user init is basically I have set up all these 4 to 0 and I have set up eip to a function called forkret alright.

And, so this is a special case where switch was not called by a function, but switch right. So, switch was not called by a function, but we just initialized it as those switch was called by a function before forkret right so that you know when it actually starts running it starts running at the function called forkret.

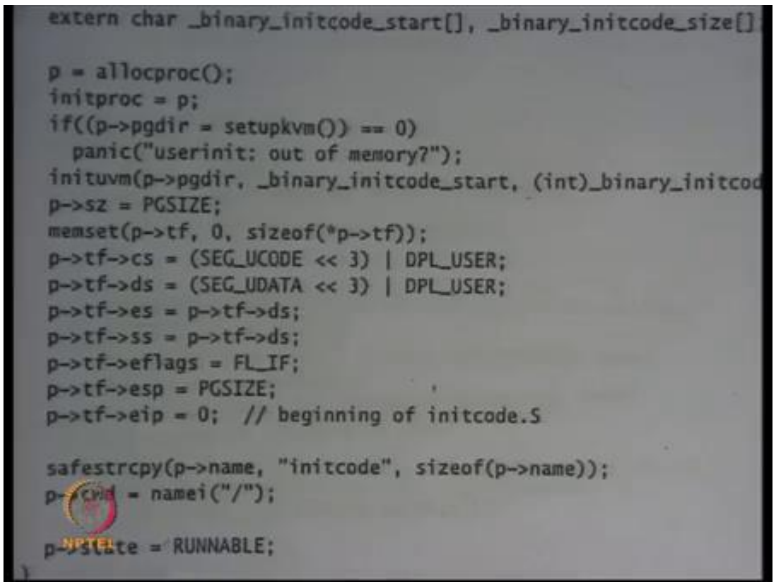
(Refer Slide Time: 20:53)



So, in this diagram basically what I have done is. So, this is the this is for a general process for anytime, but at for userinit for userinit what I have done is basically instead of any general eip I have set it up to forkret right and actually there is nothing here. So, this is both short circuited. So, the next word after forkret is trapret, if you look at how allocproc worked and then everything above it is trapframe which is completely uninitialized value at this point alright.

So, my stack is basically this large which has four registers forkret, trapret and the trapframe that is what allocproc does right. This is same the same function is called even on a fork so, that is how it initializes a new process right. So, this name forkret basically means that this does the first function you should call for the process that just started after fork. So, let us look at; let us look at them. So, first before we look at the forkret function let us look at userinit after allocproc.

(Refer Slide Time: 22:01)



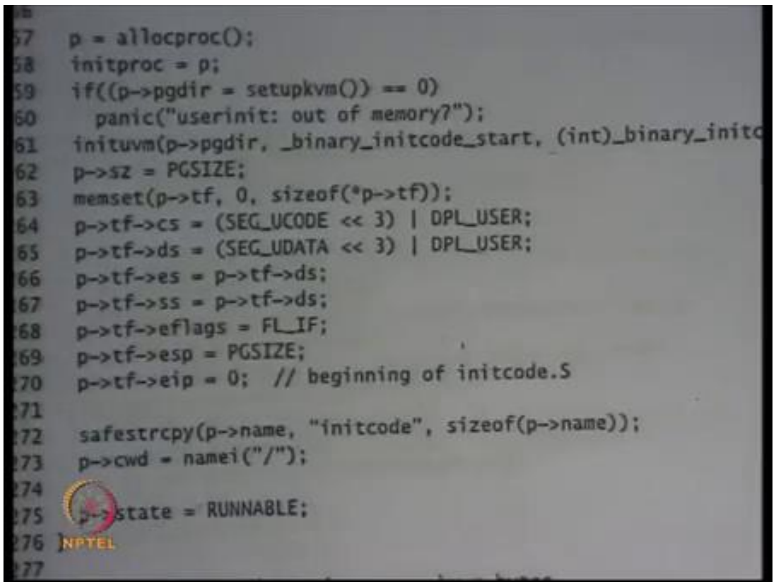
```
extern char _binary_initcode_start[], _binary_initcode_size[]

p = allocproc();
initproc = p;
if((p->pgdir = setupkvm()) == 0)
  panic("userinit: out of memory?");
inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcod
p->sz = PGSIZE;
memset(p->tf, 0, sizeof(*p->tf));
p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
p->tf->es = p->tf->ds;
p->tf->ss = p->tf->ds;
p->tf->eflags = FL_IF;
p->tf->esp = PGSIZE;
p->tf->eip = 0;  // beginning of initcode.S

safestrcpy(p->name, "initcode", sizeof(p->name));
p->cwd = namei("/");

p->state = RUNNABLE;
```

So, I have called allocproc here right and then I set up some global variable called initroc to p, let us ignore that for now and then I say p. So, allocproc did not allocate page directory right. The caller of the allocproc is supposed to initialize the page directory.

In the case of the fork you will just copy the page directory; in the case of userinit you are going to initialize the page directory. And, the first thing you have to do is initialize the page directory such that all the kernel mappings get created recall that setupkvm creates all the kernel mappings above kern base.

So, for example, that creates mappings from 0 to kern link and then you know for all the heap and then for all the many more devices on the top right. So, that is setupkvm ok. And, then it so, at this point the page directory basically has all the kernel mappings but has absolutely no mapping from 0 to kern base.

(Refer Slide Time: 23:00)



```
p = allocproc();
initproc = p;
if((p->pgdir = setupkvm()) == 0)
  panic("userinit: out of memory?");
inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initc
p->sz = PGSIZE;
memset(p->tf, 0, sizeof(*p->tf));
p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
p->tf->es = p->tf->ds;
p->tf->ss = p->tf->ds;
p->tf->eflags = FL_IF;
p->tf->esp = PGSIZE;
p->tf->eip = 0;  // beginning of initcode.S

safestrcpy(p->name, "initcode", sizeof(p->name));
p->cwd = namei("/");

p->state = RUNNABLE;
}
```

And, then there is function called inituvm that creates mappings for the user side and what is going to do is it is going to create a mapping for the addresses 0 to something see 0 actually the first process is assumed to be less than a size of a page. So, just allocate create some mapping for first page 0 to page size and fills that location with the value of this array, no binary initcode start and binary initcode size ok. So, this array is pasted at location 0 till page size ok.

So, what is he going to do? He is going to paste. So, the kernel already knows that this is the first process in want to run. So, he is going to first take that process, paste at location 0, setup eip to 0.20 the users eip in the trapframe and is going to just leave it in the scheduler and the scheduler is going to start running that process immediately, yes.

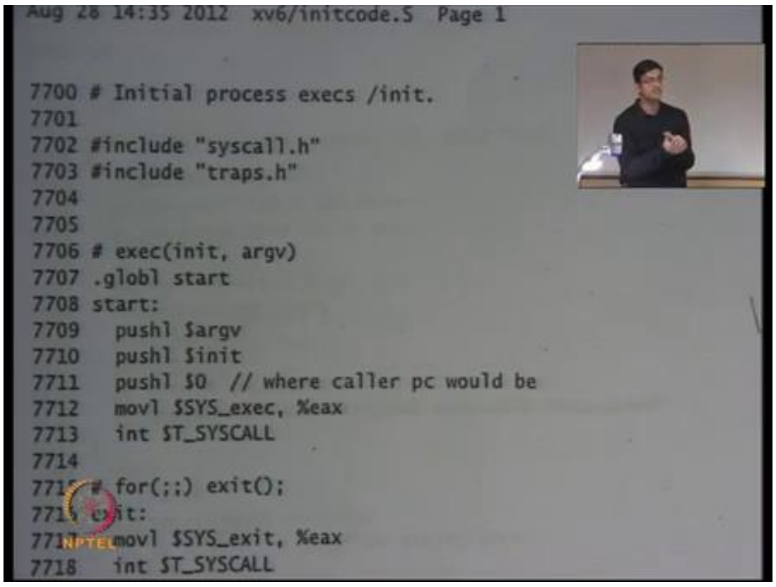
Student: Once again could you explain what binary init code start?

So, what is binary init code start? Binary init code start is some global array in the kernel. It contains the contents of this array are basically the code in that should be executed in user mode ok. So, once again binary init code start is an array and the contents of this array is the code that should execute in user mode right starting at the first location 0th byte ok.

And, so the linkers scripts have been arranged in such a way that there is some code that has been created. So, what will what the linker what the complier will do it will compile this code the code is written in assembly let us say. So, it is going to compile that code it is going to get some binary representation of that code it is going to take that code and put it in the array binary init code start. And, now this userinit function is going to take contents of that array and put it at location 0 till page size and setup my trapframe accordingly and that is my first process.

So, let us look at the code that is going to run in the beginning right. So, what are the contents of binary initcode start? The contents of binary initcode start are in this file called init dot S init code dot S that is on sheet 77.

(Refer Slide Time: 25:26)



Aug 28 14:35 2012 xv6/initcode.S Page 1

```
# Initial process execs /init.

#include "syscall.h"
#include "traps.h"


# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
```

So, that is init code dot S and these are the instructions that going to that are going to get executed when the first process actually gets to run. So, all it is doing is it is making us it is pushing an argument argv and pushing. So, these are global variables argv and init that contain the name of a file that needs to execute and ultimately just making a call system call called SYS exec. So, it is just calling the software interrupt and it is going to make a exec system call.

So, the first code is going to do nothing, but just make us in exec system call on this file name called init with the arguments called argv; and this init and argv are global variables that contains strings which are the name of the file they needs to get executed and argv is the name of the arguments anyways execute ok. So, all it is doing is making the exec system call.

So, the first process is going to make the exec system call and the assumption is of course, that by the time first process gets to run your interrupt descriptive table has been setup so, system calls can be actually executed. Make sense?

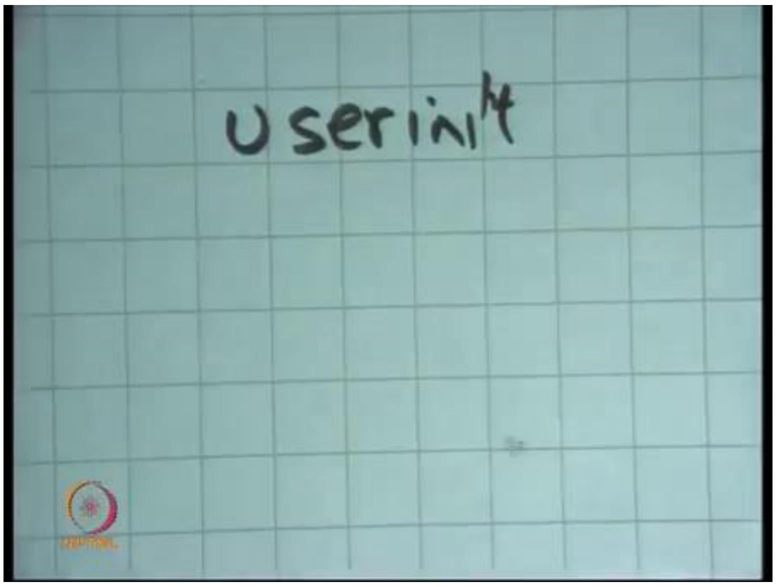
Student: (Refer Time: 26:40) any where is this files store? Actually, we do not have any file right now right so.

So, I mean this is just code that gets compiled by the compiler right, but it does not get linked to your kernel right. So, the linker is just going to look at the assembly the binary code of this particular code and it is going to paste it in this array called binary init code start how it happens you know let just ignore that.

Student: Sir, it would not execute actually?

It would not be executed right. So, the kernel will never execute this code. The kernel looks at this code as data that it puts into the address space of the first process that is all; for the kernel, this code is just data ok. So, let us look at this again let us see ok.

(Refer Slide Time: 27:31)



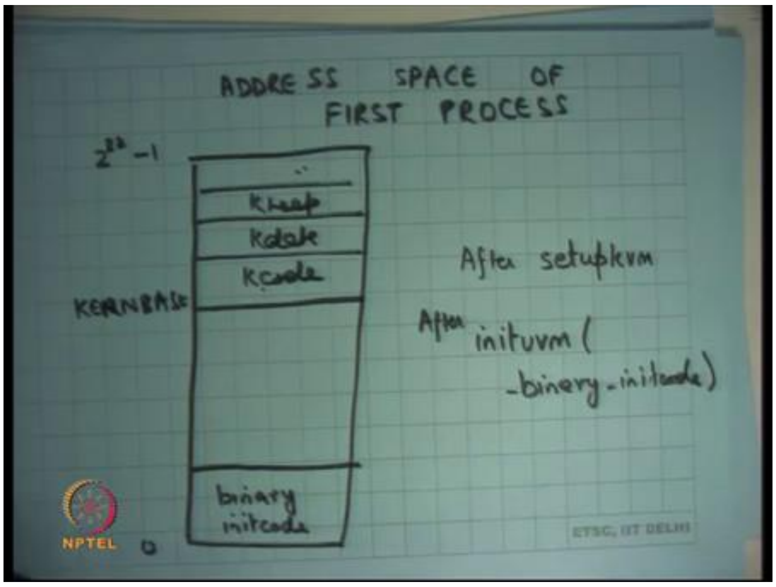
So, we are talking about userinit that is going to create the first process right. So, what is going to do we going to allocate a new PCB it is going to initialize the new page directory ok. And, it is going to paste the contents of the first program that it wants to run into this page directory into this address space and that is it.

And, I am just showing you what is the code that executes in the user mode for the first process. The code that executes for the in the user mode for the first process is just making an exec system call and then making an exit system call after that I guess right.

So, xv6 is the Unix like kernel, it implements Unix system calls. We have seen Unix system calls, you have seen fork, you have seen exec and xv6 also implements fork and exec and so, the code the first processes code is going to use this system call called exec to start running something. And, the and it is going to exec this program call slash init and init is going to execute let us say the shell or something right.

Anyways, so it is not important to know exactly how this is been done but let us just come back to userinit sheet 22. So, let us look at what happens at this point right. Let us look at what happens at this point.

(Refer Slide Time: 29:04)



So, if I were to draw the address space of first process ok. So, I am just creating the first process and I am now saying let us look at what this address space looks like let us. So, we have seen what an address space of process looks like. Well, it just looks like you know this box going from 0 to 2 to the power 32 minus 1, this is the virtual address space right.

And, I am and what after setupkvm that is kernels virtual memory the address space is going to look something like this it is just going to have a mapping from kernbase for

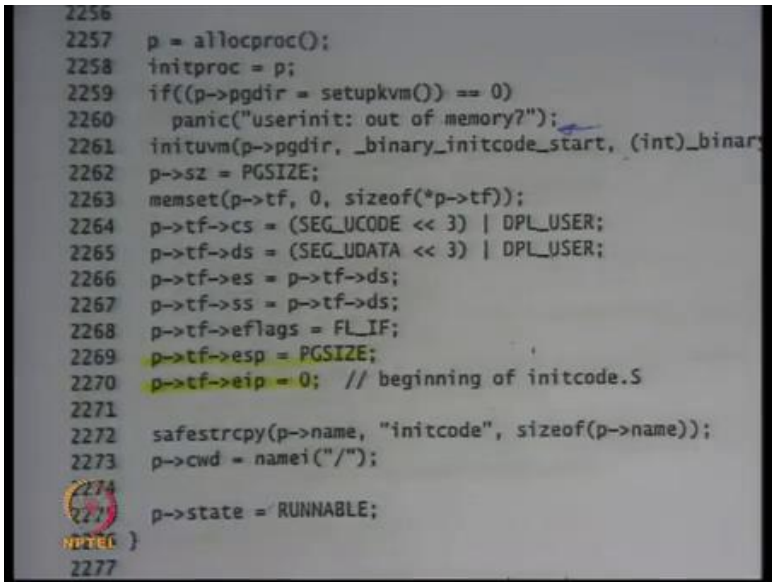
Kcode, Kdata, Kheap and so on right. So, that is what setupkvm is going to do. It is going to create all these mappings in the address space. So, this is after setupkvm.

And, then there is this function called inituvm after inituvm and inituvm basically takes arguments as an array called binary initcode. So, after inituvm binary initcode what is going to happen is, it is going to paste binary initcode here that is what it is going to do.

So, in the address space it is going to paste binary initcode here and it is going to setup the eip to 0. So, the first instruction that gets executed is whatever there was in binary initcode right. And, what we what I show just showed you is so, the first instruction that executed is just making an exec system call right.

So, what will happen is that it will the controller come here it is going to make an exec system call and the exec system call will proceed just like it is supposed to proceed which is going to going to replace these contents with whatever the contents of that file load that file into these contents ok. Is this clearer? Alright. So, this is binary initcode. So, that is what is that is what inituvm is going to do and I just showed you the code for binary initcode which just going to you know exec the first process.

(Refer Slide Time: 31:49)



```
  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binar
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;
}
```

So, as I say inituvm that is going to create that mapping I have also so, there is a field in the struct proc while size what is the size of the process. So, this first process has a size of one page that is all. So, this you know the kernel just knows that the binary initcode is less than one page so, just creates size of one page. So, size is going to page size. Finally, it is going to start initializing the trapframe right. So, I said the trapframe was left initialized by allocproc.

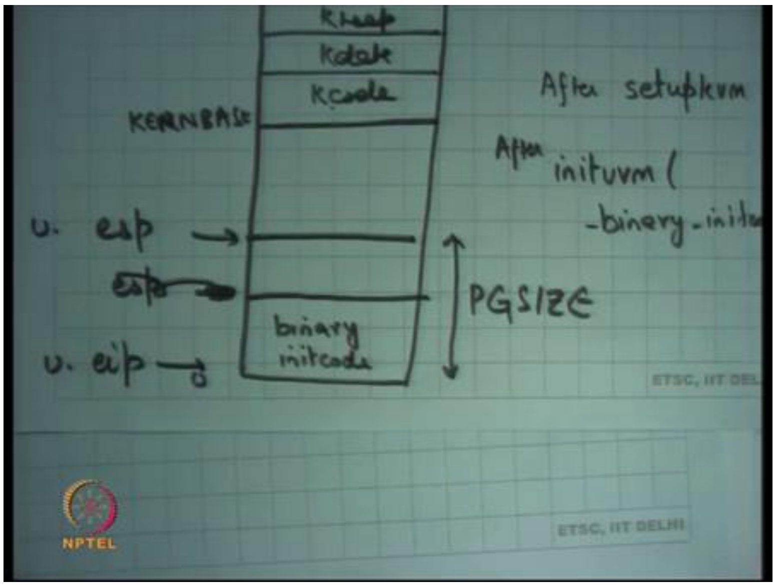
So, in this case it is going to start initializing the trapframe. So, it first 0s out all the contents of the trap trapframe. So, all the registers are 0 by default except cs is pointed to the user’s code segment right. Similarly, ds are pointers to the users data segment. Why cannot we have done if for example, I have done it to the kernels codes segment then what will happen?

Student: (Refer Time: 32:40).

So, if I had instead of SEG UCODE written SEG KCODE then when I would have transferred to the user the first instruction of binary init code would have executed in kernel mode right. You do not want that right because those programs are going to take user input which is untrusted, and you do not want. So, it is going to take user program and going to call exec in them and all that. So, if all those things start executing in kernel mode you are in bad shed.

You want that those should execute in user mode and that is what you know you are initializing the segments to user mode so that they execute in unprivileged mode. And, then you initialize other segment registers you just you know copying ds to es and fs ss eflags is initialized to FL underscore IF. What does means is the interrupt flag is enabled in the user mode right. So, that is what it means and esp is initialized to page size, what does that mean? What does that mean?

(Refer Slide Time: 33:43)



Student: (Refer Time: 33:47).

Esp will or let us say I allocated one page. So, this is page size this is page size and the first few bytes where taken by binary init code and you initialized esp here. So, binary init code needs some stack to run. So, you initialize the stack at page size, that is what he is doing and this just going to use 4 2 or 3 or 4 words in the stack to just call the exec system call. And, it is going to initialize eip to 0 that is users eip users eip and users esp right and how he is going to do it he is just going to change the values in the trapframe for registers esp and eip.

Esp is pointing is initialized to page size and eip is initializes to 0 beginning of initcode dot S. And you setup the state to be RUNNABLE; moment you set it to be RUNNABLE you know a scheduler can pick it up and start executing it. In this case I am executing the first process so, you know there is no other CPU let us say enabled, but the next thing the main is going to do is make a call to scheduler and scheduler is going to start taking processes question.

Student: Sir, is it not the address 0 that we are setting to eip invalid because generally address 0 0 is not.

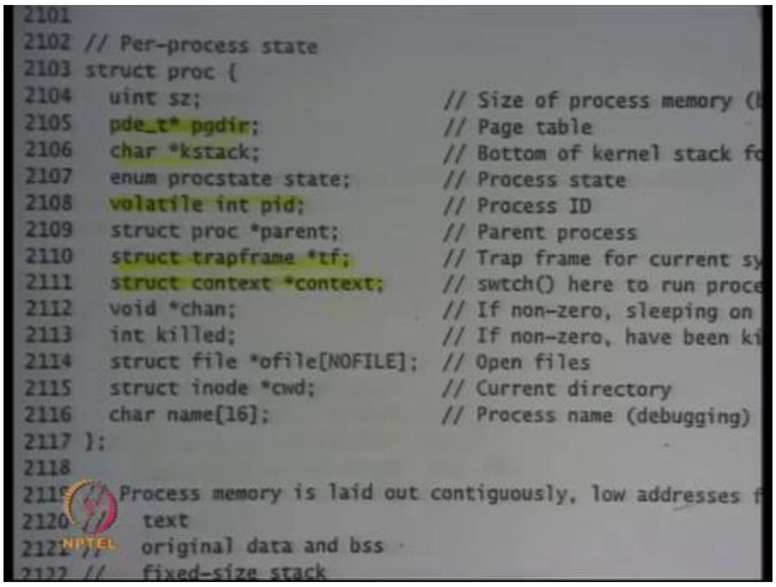
So, question is you know you know I said last time that address 0 is usually unused or unmapped so that you know you can do this. So, that you know malloc cannot return 0 as a valid value right. Well, I mean actually it is not complete so, you know my statement is not completely accurate it just that you know the this the address space is organized in a way such that malloc cannot return 0 as a valid address. So, in this case also malloc will not be able to return 0 as a valid address because you know that 0 address has the code map to it. Heap is not mapped to 0 right.

So, basically is the address 0 should not be part of the heap right that you know that is good enough. And, in any case you know this program is not going to initialize the heap or initialize malloc or free or anything or that is what. So, it is right this is just a special program that is the convention for usual programs right. In fact, even this function would have used malloc and free because you know 0 is not part of the heap even the 0 is mapped 0 is part of the code. So, it cannot be a part of the heap alright and we have the first process.

So, we initialize the page directory, we pasted some contents into the address space, we initialized the trapframe, we initialized the kstack, we initializes the trapframe, we initialized the context and we have PCB which is full, we set its state to RUNNABLE and that is it and we let the system run and the first process is going to get to run. The first process is going to call an exec that is going to you know execute some program, that program is going to let say initialize it is file descriptor and call fork depending on what command you gave on keyboard on the standard input ok.

And, so your first process is running after that right. So, just for completeness let us also look at other things in the proc structure.

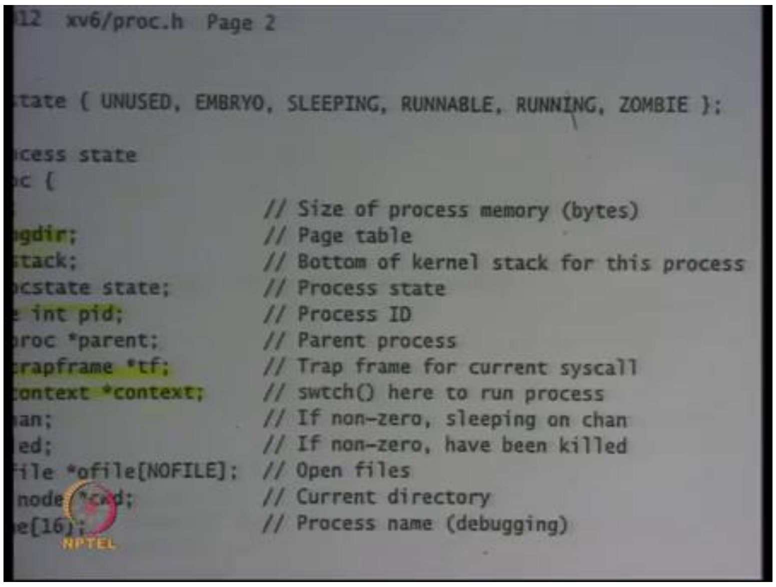
(Refer Slide Time: 37:20)



```
// Per-process state
struct proc {
  uint sz;                      // Size of process memory (b
  pde_t* pgdir;                 // Page table
  char *kstack;                 // Bottom of kernel stack fo
  enum procstate state;         // Process state
  volatile int pid;             // Process ID
  struct proc *parent;          // Parent process
  struct trapframe *tf;         // Trap frame for current sy
  struct context *context;      // swtch() here to run proce
  void *chan;                   // If non-zero, sleeping on
  int killed;                   // If non-zero, have been ki
  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 f
//   text
//   original data and bss
//   fixed-size stack
```

So, here is the proc structure. We have looked at kstack, we looked at page dir, we have looked at state right.

(Refer Slide Time: 37:31)



```
12  xv6/proc.h  Page 2

tate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

cess state
c {
                       // Size of process memory (bytes)
gdir;                  // Page table
tack;                  // Bottom of kernel stack for this process
cstate state;          // Process state
 int pid;              // Process ID
roc *parent;           // Parent process
rapframe *tf;          // Trap frame for current syscall
ontext *context;       // swtch() here to run process
an;                    // If non-zero, sleeping on chan
ed;                    // If non-zero, have been killed
ile *ofile[NOFILE];    // Open files
 node *cwd;            // Current directory
e[16];                 // Process name (debugging)
```

So, the state can be one of EMBRYO, UNUSED – UNUSED basically means this process this particular PCB is not in used because it is in array that is how you basically check whether something is used or not. EMBRYO we have seen, RUNNABLE we have seen – something that is RUNNABLE, then there is something called RUNNING. Why do I need a distinction between RUNNABLE and RUNNING?

Student: (Refer Time: 37:52).

So, if a process is already running on some CPU and then another process wants to schedule another process then this process should not be a candidate right, if a process already running then that process should not be a candidate. So, one process cannot run on two CPU simultaneously, not allowed ok. So, that is why you need a distinction between RUNNABLE and RUNNING.

ZOMBIE we know what ZOMBIE process this is. So, because xv6 does the same ZOMBIE semantics as Unix so, we it needs zombie. So, basically if a process is exited, but a parent has not called wait on it if a process is in ZOMBIE state what do you think will happen page dir will remain allocated or well should be freed?

Student: Freed, freed.

Freed. So, everything will be freed actually page dir will get freed, kstack will get freed, tf in context are meaningless you know just whatever is the status exit code of that particular process that needs to be, but I do not thing even xv6 even implements exit code so, it actually does not need anything. So, that matter right I am not sure actually.

Student: Pointer to parent also be stored.

Right and then there is a pointer to parent.

Student: Parent.

Right. So, that is again you need it for this exit rate semantics then there is a pid, parent, trapframe context we had discussed. You know let us look at channel later, killed also let us say you know let us look at later, this is an array of open files struct file, star ofile number of files maximum number of files. This is your file descriptive table that we discussed in the first few lectures right.

So, I said that every process has a file descriptive table and the process can you know call open, read, write, close, loop all the system calls. So, xv6 implements all the system calls and this is the file descriptive table with which it implements the system call and it has the same semantics that is going to start from the beginning and search for the first empty file descriptor and assign it there when you make an open call something the.

Then there is a; there is a; there is a variable called current working directory, cwd. Why do I need the current working directory? Recall that the child process on a fork in headaches the current working directory of the parent process. That is why when you type ls on the shell ls knows whose contents to display. Display the contents of the current working directory in which your parent was living right. So, that is the. So, the cwd gets copied on fork from parent to child.

And, if you ever made if you ever want to find out what where am I you just look at your you just say you know look at proc dot cwd. So, if you if ls wants to look at the current working directory it can do that alright. And, then you know process name that is for debugging process. So, that is basically yours struct proc ok.

Then, let us look at the main function once again.

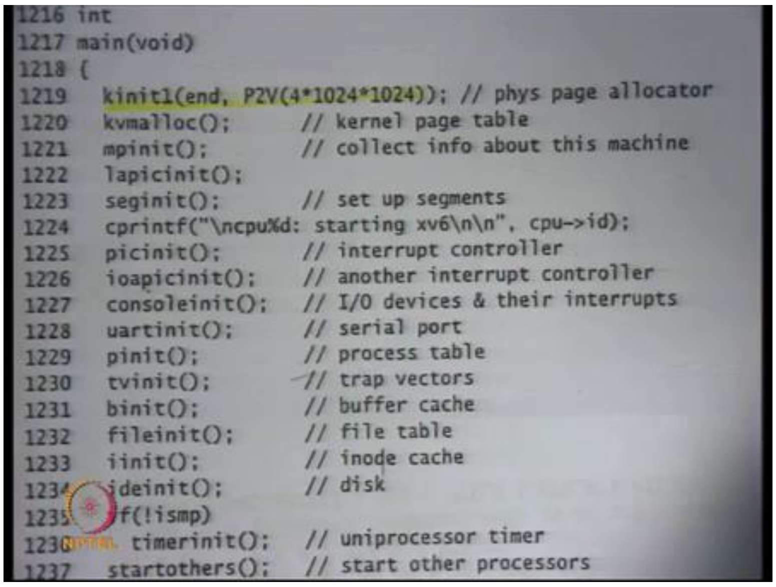
Student: Sir.

Yes.

Student: Why the type volatile in pid?

Why the type volatile? What does volatile mean I am going to discuss that later. Let us just know let just skip that for now want to discuss that later.

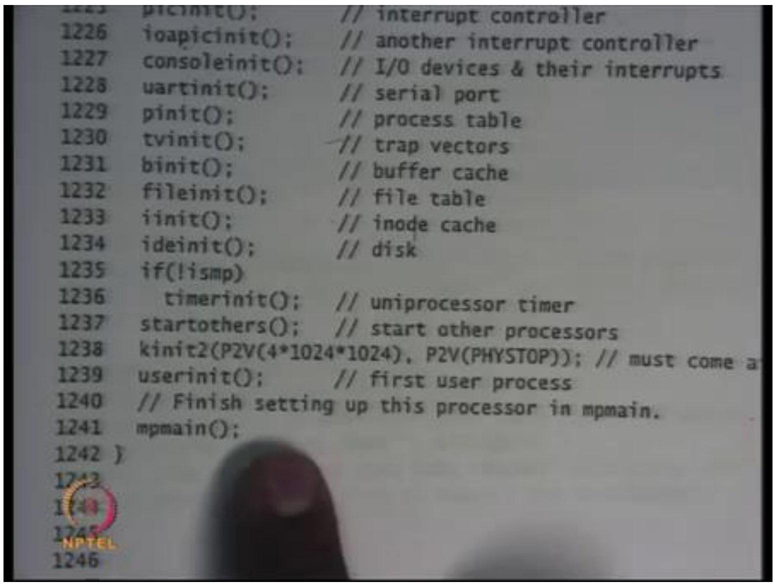
(Refer Slide Time: 41:08)



```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // collect info about this machine
  lapicinit();
  seginit();       // set up segments
  cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
  picinit();       // interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // I/O devices & their interrupts
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  iinit();         // inode cache
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
```

And, this is the main function recall that we have a we had already seen how you know transfer control transfers to main. And, then this was the heap got in getting initialized to first 4MB, the page kernel page table getting initialized then everything starts getting initialized.

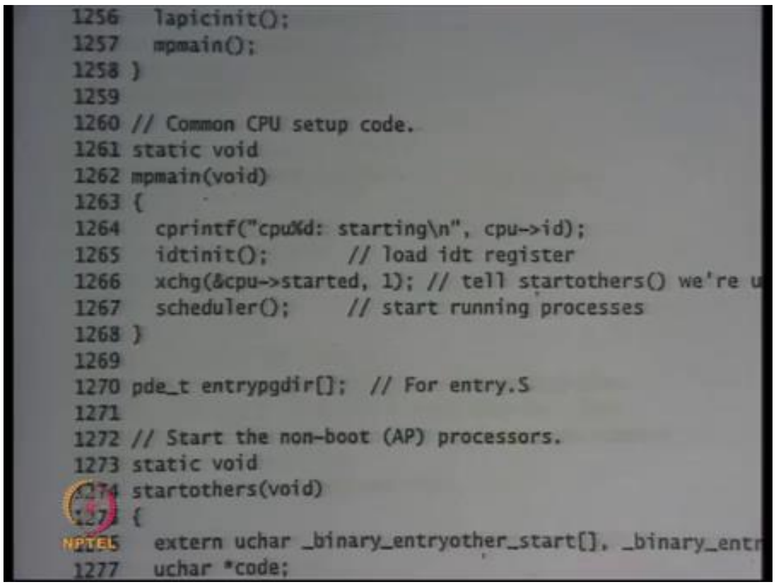
(Refer Slide Time: 41:24)



```
picinit();       // interrupt controller
ioapicinit();    // another interrupt controller
consoleinit();   // I/O devices & their interrupts
uartinit();      // serial port
pinit();         // process table
tvinit();        // trap vectors
binit();         // buffer cache
fileinit();      // file table
iinit();         // inode cache
ideinit();       // disk
if(!ismp)
  timerinit();   // uniprocessor timer
startothers();   // start other processors
kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come a
userinit();      // first user process
// Finish setting up this processor in mpmain.
mpmain();
}
```

And, at this point you initialize the heap to till PHYSTOP and this was userinit. So, as soon as you initialize the heap to PHYSTOP you called userinit so, the first process gets created and after that you call this function called mpmain, what let us say multi processor main which let us say other processes also going to call.

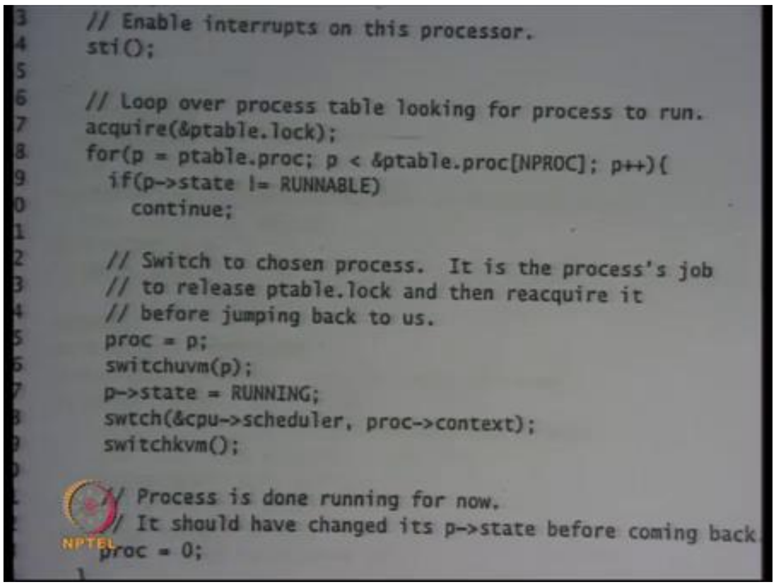
(Refer Slide Time: 41:42)



```
  lapicinit();
  mpmain();
}

// Common CPU setup code.
static void
mpmain(void)
{
  cprintf("cpu%d: starting\n", cpu->id);
  idtinit();       // load idt register
  xchg(&cpu->started, 1); // tell startothers() we're u
  scheduler();     // start running processes
}

pde_t entrypgdir[];  // For entry.S

// Start the non-boot (AP) processors.
static void
startothers(void)
{
  extern uchar _binary_entryother_start[], _binary_entr
  uchar *code;
```

What multi processor main does is it does initialize the idt right. So, just loads the idt into the idtr and then call the scheduler right. So, notice that you know in this case the idt was initialized after the first process was initialized. It is because the first process was initialized, but it would not only run after the idt we can have been initialized right and so, idt gets initialized and then you call the scheduler. And, the scheduler is going to go through the process table pick up one which is RUNNABLE and call switch to it. So, let us look at the scheduler ok.

(Refer Slide Time: 42:20)



```
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back
      proc = 0;
    }
```

So, that is sheet 24 that is the scheduler. And, scheduler is running on what stack? The scheduler is running on the same stack on which main was running need. So, there is just one initial stack on which another scheduler is running because you just made a function call to scheduler. But what it is going to do is it is going to let us ignore the locking, but it is going to go where the process table and it is going to find if.

So, if it is not RUNNABLE then you ignore it, but if it is RUNNABLE then you fall through, and you call this function called switchuvm to p right. So, what is switchuvm going to do? Firstly, it is going to load the p's page table into cr3. No problem because p's page table also has mapping for this kernel so, you can just switch from. So, switch you may has changed the address space right there alright.

So, it has changed the address space it will also set the stack initialize the stack pointer into tss segment right. So, it will also initialize the stack pointer of the new process. So, it will take the esp value of the new process and put it in the tss segment.

So, when you are going to transfer control to this process and there was an interrupt then the right stack gets loaded. You should not load because right now you are executing on the kernel on the schedulers stack, now you want to execute on the process stack it is. So, it is also going into load the value into tss, so that next time there is an interrupt it comes into that stack.

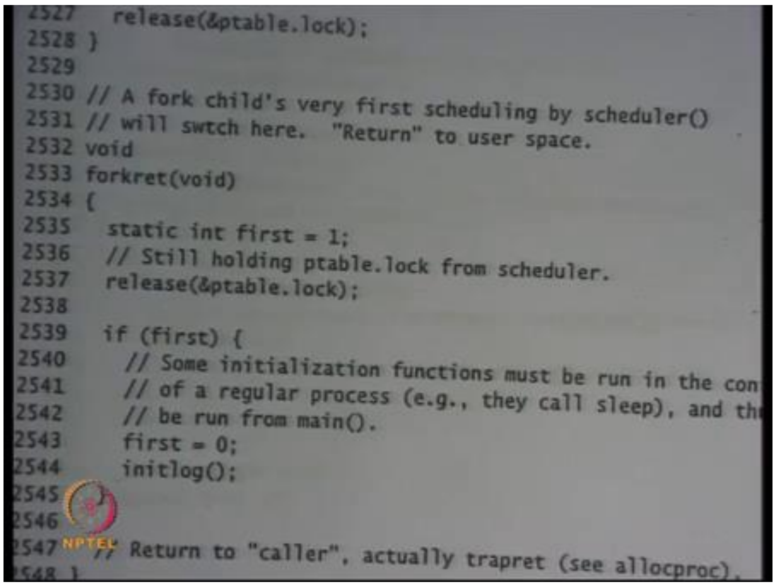
Setup the state to RUNNING and then call this function call switch that we have seen last time and the switch is going to switch from the schedulers stack to that processors stack and actually it is not never going to return. It is not going to return immediately; it is going to return at the next switch. So, the scheduler call switch, but never returns from here right, but does not return immediately at least.

The process stack has been loaded the process stack returns. So, the function that will called after this is in the case of first process what is going to get called?

Student: Forkret.

Forkret right. So, immediately after this forkret gets called; you do not return here, but it is forkret that call and it is been it is called on the new stack with the new stack been loaded into the tss and with the new address space.

(Refer Slide Time: 44:52)



```
2527   release(&ptable.lock);
2528 }
2529
2530 // A fork child's very first scheduling by scheduler()
2531 // will swtch here.  "Return" to user space.
2532 void
2533 forkret(void)
2534 {
2535   static int first = 1;
2536   // Still holding ptable.lock from scheduler.
2537   release(&ptable.lock);
2538
2539   if (first) {
2540     // Some initialization functions must be run in the con
2541     // of a regular process (e.g., they call sleep), and th
2542     // be run from main().
2543     first = 0;
2544     initlog();
2545
2546
2547   // Return to "caller", actually trapret (see allocproc).
2548 }
```

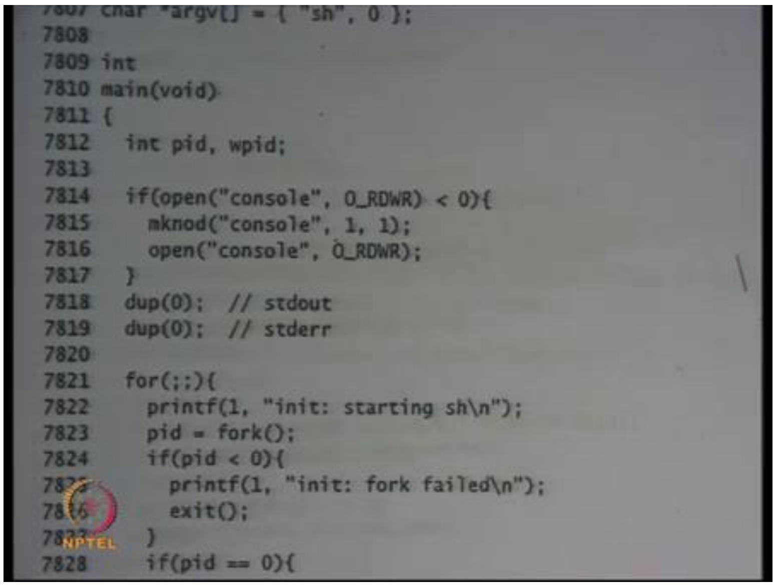
Let us look at forkret sheet 25 well forkret does nothing. So, this is line 2534 is where it is going to start executing and you know this is just global variable you know static variable. So, let us ignore this it just releases some lock basically. So, let us ignore that also it initializes somethings and then it returns to the caller. For the first process who is the caller of forkret?

Student: Trapret.

Trapret right. So, as soon as it calls return here it is going to get back to trapret. Recall that the stack was setup in such a way that it looks like it was the trapret called forkret right because you had traprets eip and then you had forkrets eip. So, now forkret gets to run. So, forkret could have pushed something on stack, but it also its responsible to pop those things from stack calling conventions and then it is going to call ret so, it is going to return to trapret.

Fork and trapret is going to do what? It is going to pop off the trapframe and return to user mode and trapframe was initialized by a userinit function, so, great. We are running our first processes ok. So, finally, let us just look at the init code function on sheet 77 so which we skipped.

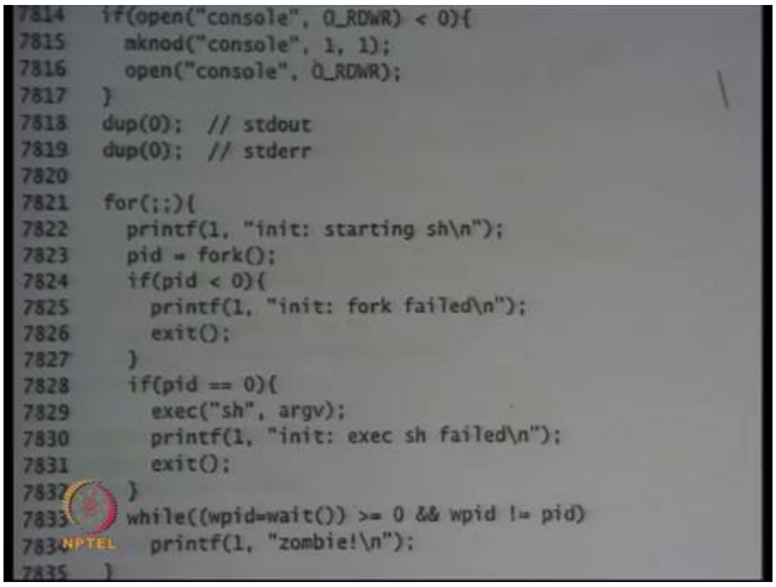
(Refer Slide Time: 46:05)



```
char *argv[] = { "sh", 0 };

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
```

So, let us say it is just going to make a system call called SYS exec and it is going to call the actually init dot C and init dot C is just going to initialize its standard output and standard error. It is also going to initialize the standard input and it is going to execute the shell inside. Let us initialize the standard input, standard output, and standard error and then it starts running the shell. The shell just takes in a command.

(Refer Slide Time: 46:34)



```
  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
```

And, so here it is just calling exec sh and sh is going to take in command and start executing it right. So, that was; so, that is how you get a shell when you run the xv6.

Let us stop here.