

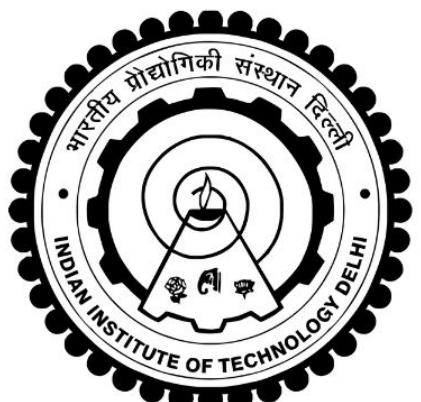


Operating Systems

Prof. Sorav Bansal

Computer Science and Engineering

IIT Delhi



INDEX

S. No	Topic	Page No
1	Introductio to UNIX System Calls Part - 1	1
2	Introductio to UNIX System Calls Part - 2	20
3	Threads, Address Spaces, Filesystem Devices	46
4	PC Architecture	70
5	x86 Instruction Set, GCC Calling Conventions	95
6	Physical Memory Map, I/O, Segmentation	118
7	Segmentation, Trap Handling	142
8	Traps, Trap Handlers	164
9	Kernel Data Structures, Memory Management	189
10	Segmentation Review, Introduction to Paging	216
11	Paging	237
12	Process Address Spaces Using Paging	262
13	Translation Lookaside Buffer, Large Pages, Boot Sector	284
14	Loading the kernel, Initializing the Page table	311
15	Setting up page tables for user processes	336
16	Processes in action	360
17	Process structure, Context Switching Process Kernel stack, Scheduler, Fork,Context-Switch, Process Control Block, Trap	384
18	Entry and Return	408
19	Creating the first process	432
20	Handling User Pointers, Concurrency	462
21	Locking	485
22	Fine-grained Locking and its challenges	511
23	Locking variations	538
24	Condition variables	566
25	Multiple producer, multiple consumer queue; semaphores; monitors	592
26	Transcations and lock-free primitives read/write locks	619
27	Synchronization in xv6: acquire/release, sleep/wakeup, exit/wait More synchronization in xv6: kill, IDE device driver; introducion to Demand	641
28	Paging	668

29	Demand Paging; Introduction to Page Replacement	699
30	Page Replacement, Thrashing	725
31	Storage Devices, Filesystem Intrerfaces	750
32	File System Implementation	775
33	File System Operation	802
34	Cash Recovery and Logging	827
35	Logging in Linux ext3 filesystem	850
36	Protection and Security	876
37	Scheduling Policies	902
38	Lock-free multiprocessor coordination, Read-Copy-Update	929
39	Microkernel, Exokernel, Multikernel	955
40	Virtualization, Cloud Computing, Technology Trends	978

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

Lecture - 01
Introduction to UNIX System Calls
Part -1

Welcome to Operating Systems lecture 1. This is an introductory lecture and we will basically in this course discuss about operating systems and how they work and what are the issues when you design an operating system and implement one. So firstly, why is operating systems an interesting thing to learn ? Well, if you look at the history of computing, most of the landmark events in the history of computing have involved operating system either at the center or at the as one of the components of the event.

So, right from 1960's till 2014 operating systems have gone through series of design changes, implementation improvements, and has always been an active area of research. So, to give you a flavor, in the early days of computing an operating system used to control large machines; for example, the IBM mainframes and allowed people to run multiple processes and allow multiple users to share that large machine and get their work done. Today operating systems are present in your personal computers, they are present in your mobile phones, they are present in your cars and in all kinds of embedded devices .

The lots of different open problems that operating system researchers have to face today; for example we are seeing rapid changes in our hardware. We are going from high performance single core computers to computers which have lots of cores. Already are small machines like laptops have up to 4 cores or 8 cores and large machines, larger machines can have up to 80 or 100 cores, that is not uncommon.

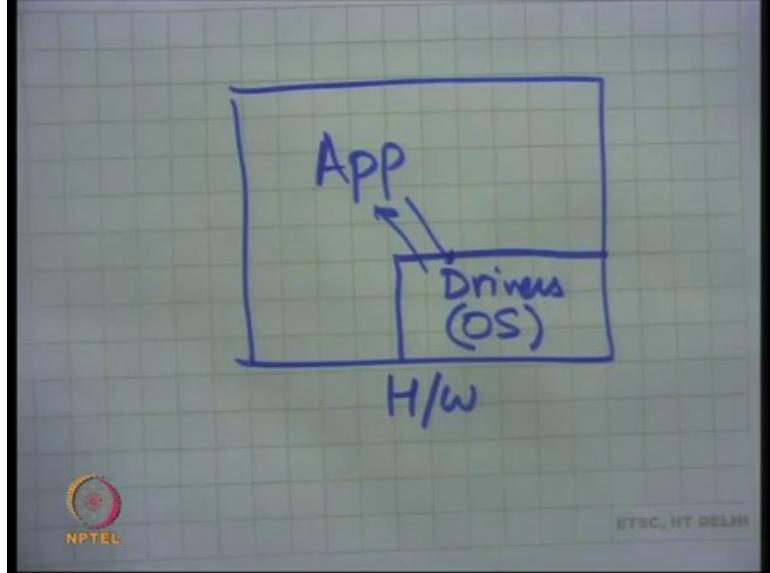
And so, researchers are constantly asking the question, what is the right operating system for this new kind of hardware and do or of current operating systems really fit for these kind of new devices. Similarly we are seeing, advent of new kinds of devices which need operating systems like phones; will the operating system that runs on your desktop, be the ideal operating system for your phone or something better is needed.

Then there are reliability issues; the operating system that runs inside a space mission, let us say the mars mission or something has to be much more reliable than your desktop operating system. And so, reliability is an operating system issue and it is a constant effort of researchers to ensure that operating systems do not have bugs or have ways to prove that operating systems do not have work, so all these are also research issues.

Performance is definitely an operating system issue, I want to write an application and the application is going to run on a certain set of hardware; which perhaps includes multiple multiple processors, each having multiple CPU cores and having memory, disk, perhaps accelerator devices like the GPU's the graphic processing units etc. And how can the operating system allow applications to use all this hardware in the most efficient way. And all these are basically very very interesting software engineering, software design and in particular operating system design problems. So, let us understand what an operating system really is. So, and we will we are going to look and try to understand this by looking at the history of operating systems.

So, the first operating system was perhaps not designed, it was not very ambitious in its goals; where the goal of an operating system was to allow a program to run. And so, in some sense operating system is the lowest layer of hardware a software that sits on top of hardware. And so, the first operating system would just export a certain set of libraries, that will allow an application to use this hardware. So, the picture would perhaps look something like this.

(Refer Slide Time: 05:06)



You have let us say if I draw the whole system as this box and hardware as the outside area of the box, then the operating system is basically a bunch of device drivers. And the application is running on top of this operating system and making calls into the operating system to access the hardware. So, the application runs on top of hardware, but for some operations it can ask the operating system to do it. And so, different applications need to be written for this operating system in this way.

Notice that in this picture, I am only drawing one application running at one time. So, there is only one application that is running at any time on this operating system. And so, this is a uniprocessor operating system or uni process, that's why also called uni processing operating system. And so, only one application can run it at any time and the operating system is just a collection of libraries and which is allowing the application to access the hardware. Let us take a step back and let us also understand how a computer system should really get organized. So, one way to organize an operating a computer system or a software system is to just write all your software as one big program.

So, let us imagine a world where your computer, still the hardware is the same; which basically means there is a processor, there is memory, there is disk etc and they are all interconnected just the way they are today. But the software was indeed written in a one monolithic style, so there is one large program; that is going to do all your things for you. So, for example, this program contains the logic to boot your computer from power of states to something which is usable; it contains all the logic to implement your editor, your shell, your GUI the graphics window manager, your browser, web clients, web servers and all this.

So, this program could be one large program that has all these things built into it. And so, there is this program would probably contain one large case statement, which says if the user does this then go here, if he presses a button here then go there, if he types this command then jump there and so on.

So, you can imagine that all your software on your computer can be organized as one big program that has all these things built into it , it is fact, it is possible; it is not really practical. Because completely different pieces of logic, an MP 3 encoder has nothing to do with the web server, have to be part of the same program and the developers of these two different logics have to now talk to each other, so that they are compatible with each other and can fit inside the same program. Also if there is an update to one part of the program, it basically means updating the entire program.

Moreover there are trust issues; what if I am I want to implement a web server, but I also know that I am sharing this my I will be a part of this large program that also runs the MP 3 encoder. But I do not trust the developer of the MP 3 encoder, because perhaps I do not trust his I do not trust his program; because perhaps here I know that his program may have bugs.

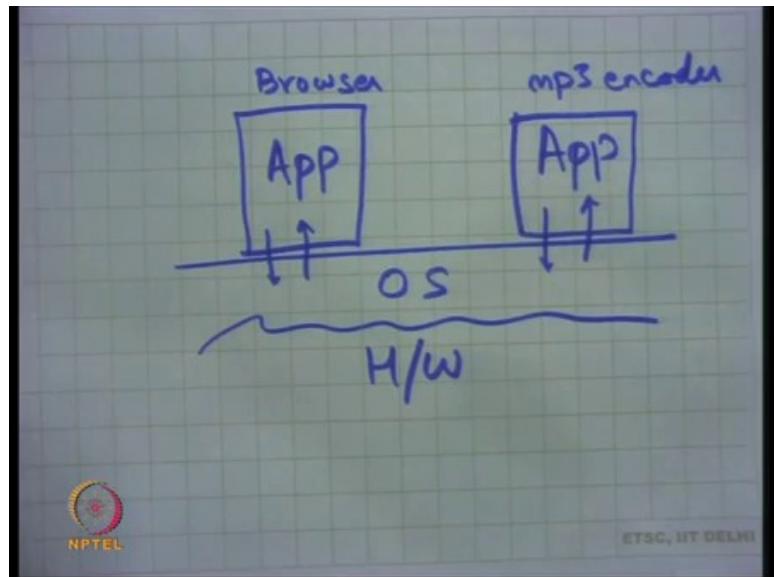
Or I just do not trust the developer basically; that means, that I do not trust him and I do not want to trust him, basically maybe he may want to do malicious things to me. So, all these things are completely not possible in this model where you have one large program. And of course, it has it is also a software engineering nightmare, because if you have one large program that all these things how is it possible to maintain this over a long period of time.

So, in theory there could be an operating system that just implements all the functionality inside it, as one large program and do things. But that is not going to be very practical. So, typically what operating systems do is expose an interface. So, which allow applications to run on top of the operating system. And these applications can be implemented independently.

So, one application could be the MP 3 encoder for example, and the other application could be the web server; and these applications can run can be implemented independently. And at the onetime let us say one application or multiple applications can run together, they all rely on the same interface that the operating system provides. And so, these applications can run on this operating system as long as they obey the interface. In the above particular picture the interfaces happen to be the device drivers.

In yet another kind of interface, you may want that let us say there is hardware, running with my operating system; the operating system exposes certain interfaces and allows multiple applications to coexist at the same time. This is a multi-processing operating system. So, for example, this application could be a browser and this particular application could be let us say an MP 3 encoder or any other such applications.

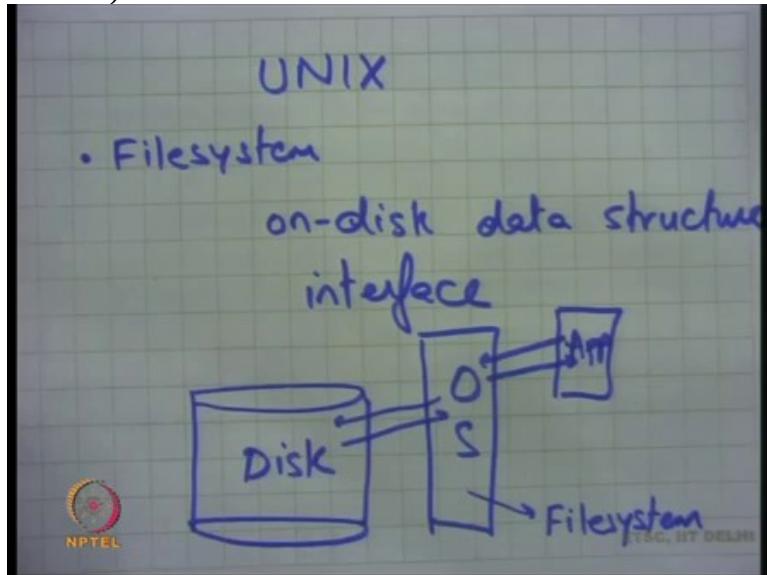
(Refer Slide Time: 10:06)



And now these two, the developers of these two applications do not need to trust each other, they also do not need to coordinate with each other. As long as they meet the specification of the operating system, they can run together at the same time without having to worry about each other. So, this kind of an architecture is much better for software from a software engineering point of view, it is also much better from a security point of view, modularity and so on and also performance.

So, let us understand what kind of, what are these interfaces, what should these interfaces look like and that itself is when it turns out is a non-trivial problem. So, what should the operating system interface be and let us again trace back to history. So, one of the first operating systems was Multics that was coming out of Bell Labs. And one of the successors of Multics was Unix.

(Refer Slide Time: 11:42)



So, there was a system called Unix as developed by Ken Thompson, who was one during award for this for his work on Unix and other people that were involved in this kind of work Dennis Ritchie and others. So, what was Unix? Well, the first version of Unix or the early versions of Unix look something like this, they said I want to be able to run multiple processes on my system and what should the operating system provide as a minimal sort of interface.

So, they started with first thinking about what. Firstly, what are the hardware components, well the hardware components are there is a processor which we call the CPU, there is memory which is RAM and then there is disk. Disk has the semantics that its contents are preserved across power reboots, also the disk needs to be shared across multiple processes . So, one process may want to access file a and another process may want to file access file b, they may be running at different times; but the files a and b need to coexist on the disk.

On the other hand at that time, memory could be assumed to be exclusive. So, you could assume that if process a is running, then only process a is using the memory and nobody else is using that memory. Notice that for the disk that is not true . For the disk if , you at the same time it is important that all the contents of all the different processes even if the process is not using them exist.

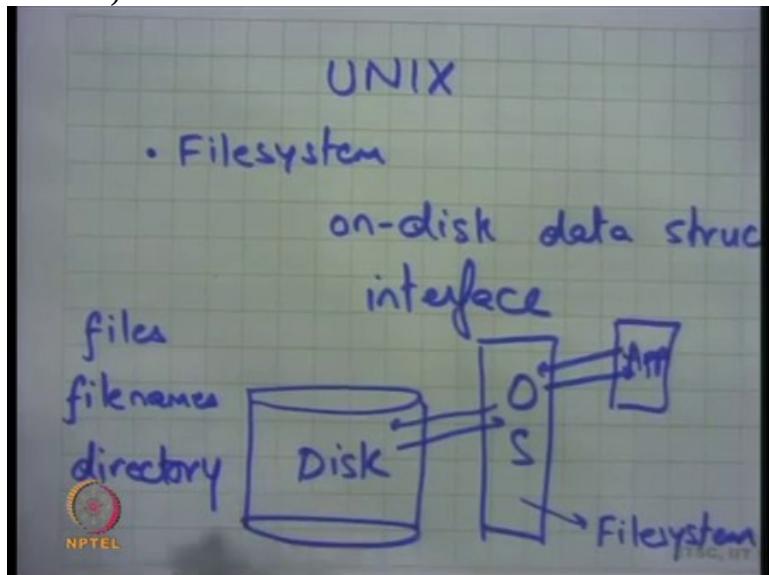
But for memory which is which whose contents are volatile; which means they do not persist across power reboots, at that time they assumed that a process basically has exclusive access to all memory . Of course, today we also do not do that, we basically allow the memory to get shared across multiple processes at the same time.

Well, let us first understand how initially the Unix interfaces look like. So, the other thing that was important was; one important program that was needed was an interactive shell . What is an interactive shell? An interactive shell will give you a command prompt, you will type in your command and depending on the command some program will get to run . When that program finishes you will come back to the interactive shell and this loop and this mechanism will continue forever; that is the minimum that a usable computer system should have.

So, the couple of things that are most very important; firstly, there should have been a file system . So, the early developers of Unix said there needs to be a file system. A file system is an on-disk data structure and some kind of interface to access this on-disk data structure . And this disk has some contents, so there is an operating system that sits between the application and the disk.

And the application is going to make some requests to the operating system and the operating system is going to translate those requests to disk request and then serve the application using those interfaces. And this translation layer was also called the file system. And so, it is clear that any operating system needs to have a file system .

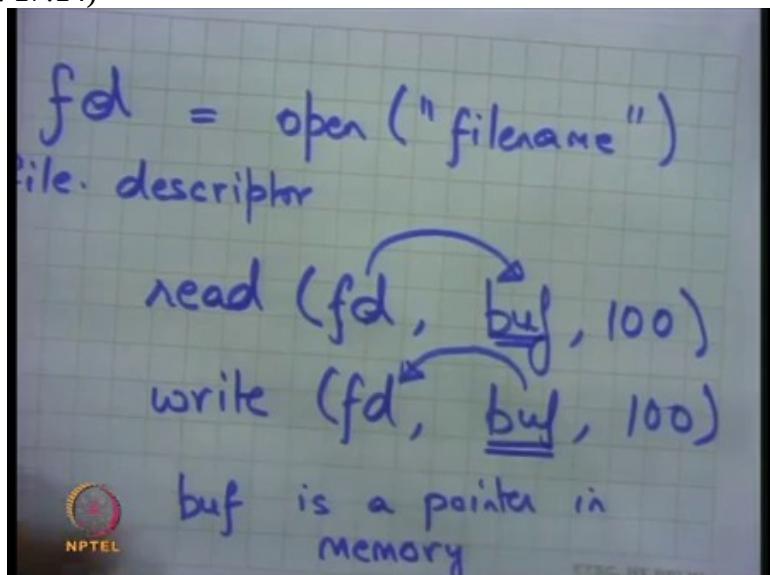
(Refer Slide Time: 18:48)



So, well the file system in early Unix look very quite similar to what we have today; which basically means that files there was a notion of files, which were nothing but streams of characters . And there were a notion of file names . And a process could say, I want to access file name a at offset b and so, the operating system will translate the file name into a disk offset and add offset b to it and give you the contents of that particular file.

Also for better manageability, the earlier file system also had the notion of directories; which basically meant that the files, the file names or the file system or the namespace of the data structure was organized in hierarchical manner. Which basically means that file names were basically had a full path name associated with it; which basically meant where do you go from and so on, which included starting from the immediate parent directory to it is parent to it is parent till you reach the root and so on ok. So, that is one thing that Unix had to have.

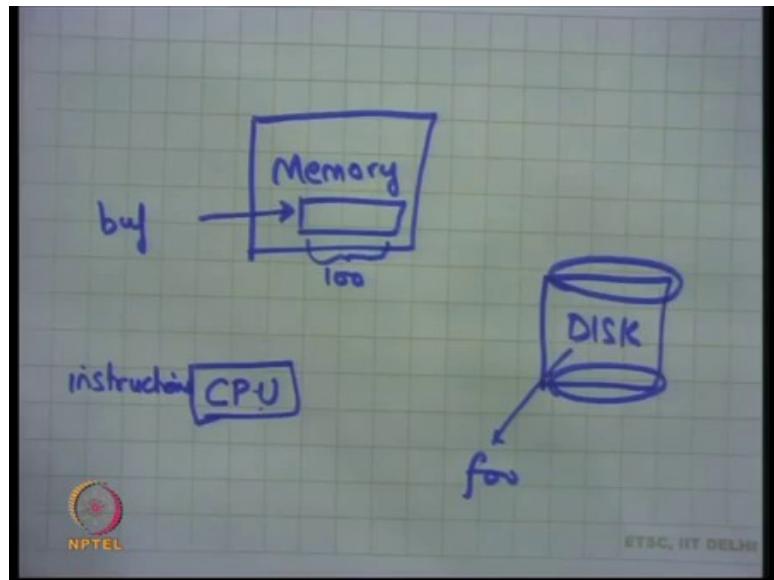
(Refer Slide Time: 17:14)



And the way they did this was basically using an interface which said you open a file; which basically means that I can say I want to open a file foo, then I basically can now use, open up. when I open a file I could let us say open file name and this would give me what is called a handler or a file descriptor which I am calling fd. Then I could read on the file descriptor, I could say I want to read in this file and I want to read from this file, which I have opened previously into a character array buf 100 bytes let us say.

That basically means that I want to read 100 bytes from this file and store those contents into buf. So, or I could say I want to write to this file a 100 characters from buf. So, read basically says, read from file and put it into this character array and write means read from this character array and write to this file. By the way what is buf? Buf is a pointer in memory.

(Refer Slide Time: 18:40)

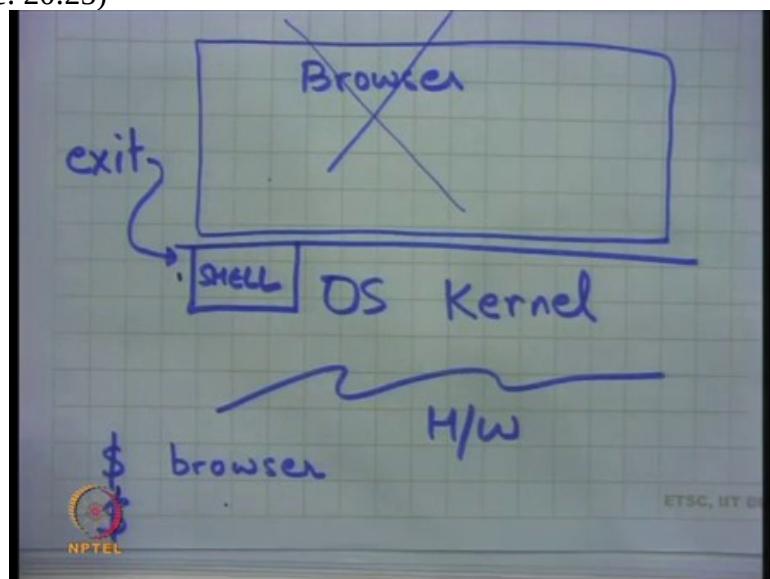


So, if I look at it again, my hardware looks something like this; I have a CPU on which my instructions run, there are memory and disk. So, the CPU is going to run this instruction which will basically make the system, which call this function called read. Buf will be somewhere in the memory and it will have let us say a 100 characters in it and the file foo will be somewhere in the disk and CPU could execute this command called read or write to transfer contents from memory to disk and from disk to memory using these read and write commands.

Notice that the application does not need to worry about which disk it is or how do I access the disk etc; all those things are abstracted away from by the operating system. The operating system knows what disk it is, how to run that disk, how to write to that disk etc. And the interface that the operating system provided or the Unix provided was this read and write calls that the application could make to read or write from the disk. that is one thing.

The other abstraction that they had was that our shell. So, they said ok, because the shell or the interactive shell has to be such an important part of the operating system, the shell was implemented inside the kernel.

(Refer Slide Time: 20:25)



So, let us say there is the operating system OS. And I am going to the operating system is also often called the operating system kernel . So, I am going to use that words kernel and operating system interchangeably and let us say there is the hardware once again.

And one part of the operating system would implement a program called shell . And let us see what the shell does; the shell basically gives you a command prompt, let us say the command prompt is dollar and then you type a command, let us say you type a command browser . So, what the shell is going to do, is it is going to check treat this command as a file name, it is going to search for this file name in the current directory where the shell is running; and if it finds a file name, then it is going to treat that file as an executable program and we are going to run it ok.

So, let us say a browser was a file that existed in the current directory in which the shell is running, then that file will get loaded as an application and control will get transferred to the browser. So, notice that basically the operating system is basically providing interfaces for you to run different programs and allowing different programs to co-exist simultaneously on the disk .

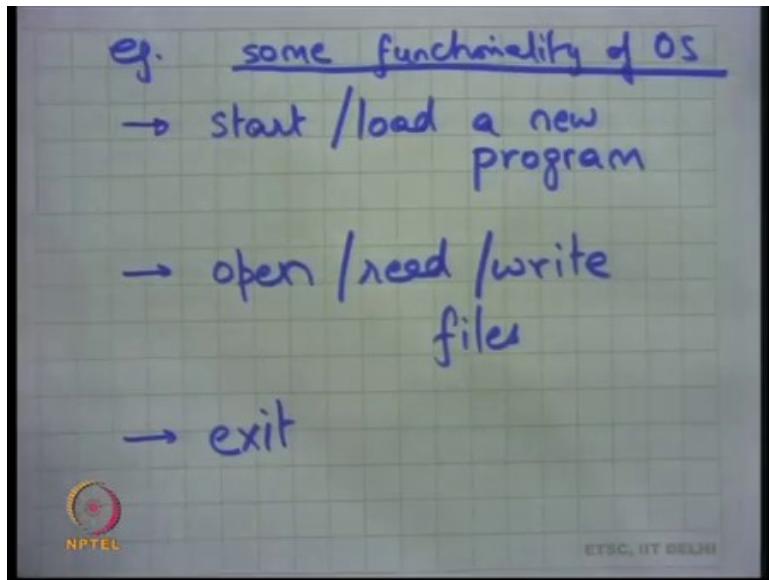
At this point we assuming that, only one program is present in memory at any time . So, and there is a special program called shell inside the operating system or the operating system kernel and this shell is going to take a command from the user; which basically mean it is going to read from the some port let us say the keyboard, so it is going to read from the keyboard, interpret that command; which basically means it is going to typically it will interpret the command as a file name, it searches for the file name in the disk in the file system really . And if it finds a file with that name, then it loads that file name, so that file basically should contain some data, some instructions that need to get executed. So, those instructions get loaded into memory and control is transferred to that particular file program; that program is now going to run all by itself .

So, it is as though nothing else is present in memory, it is just that program that is present. And so, that program is going to run all by itself and when it is running it may make more open calls to open more files; it may make more read calls or write calls to to read or write files and at some point it may want to say I am done, I want to exit .

So, what was how was exit implemented in Unix . So, there needs to be something called exit in early Unix. So, how do you exit? Well, at that time, so in the early version of Unix, exit was basically implemented by just returning back to the shell. so there was another function that the operating system kernel provided which was exit and what that will do is it will remove the occupied process from memory and jump back to the shell to take the next command .

So, you are the print the next dollar sign and here you are and you can now print your next command. There were other things that the operating system had to be careful about. Firstly, if the browser had opened certain files, then when you could return back to the shell; the first thing the shell would do is close all the open files, so that if the new program gets to run, he can open more files and so on . So, this was the simplistic model of the Unix operating system in it is early days.

(Refer Slide Time: 24:44)

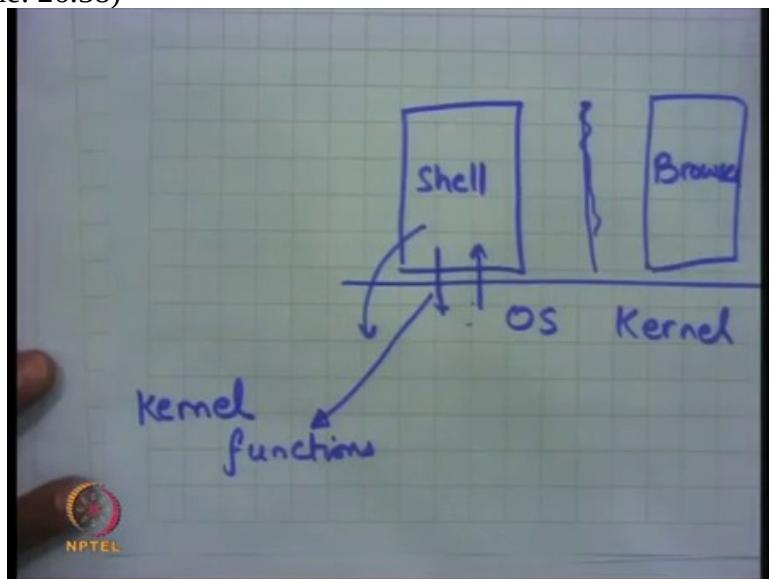


And basically what we have seen is there are few things that the operating system was providing us. Number one, it allowed us to start a new program, start/load a new program and this it was being done using the special program called shell inside the operating system. It allowed us to open, read and write files and it allows us to exit. So, some functionality of OS that applications use examples. These are example functionality that the application is using and you seen how they are using it .

So, notice that in doing this operating system design, the designer has basically carefully decided that some part of some functionality needs to be part of the operating system kernel. For example, the shell program is part of the operating system kernel and the device drivers are part of the operating system kernel. And some part of the logic does not need to be part of the operating system kernel and it should be present as application logic in executable files; that can be executed by the user at will.

Even very soon people realized that, even the shell has no need to be part of the operating system kernel. One of the important things that go into an operating system design is to make the interface as small as possible and as usable as possible and yet as powerful .

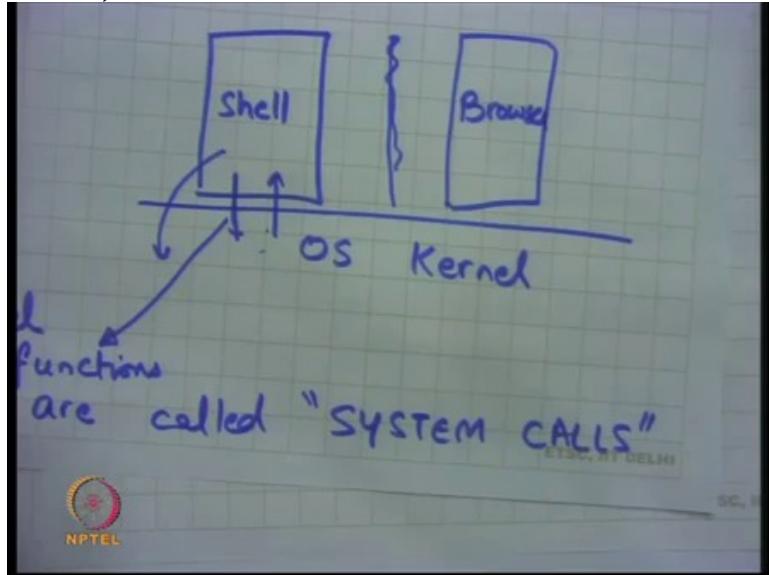
(Refer Slide Time: 26:38)



So, today the operating system kernel does not provide the shell command, instead the shell itself runs as a separate application . And the shell has ways to tell the operating system to start another application. So, for example, the shell could tell the operating system to suspend itself and start a browser; just like before except that this time the shell is not part of the operating system, the shell itself is running as an application.

And in order to do that, it is important that you all have interfaces that allow an application to be able to create another application and jump to it. So, these functions that the operating system provides; these are these functions basically form the interface of the operating system.

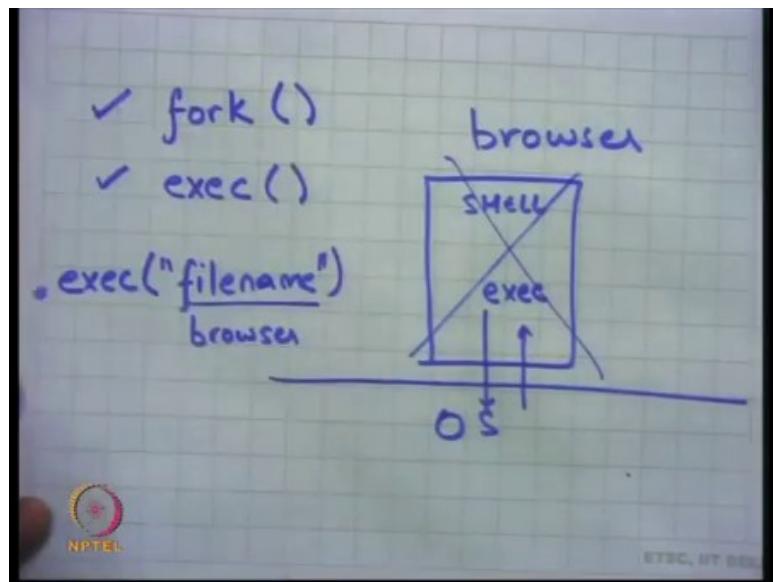
(Refer Slide Time: 27:47)



And let us call these the kernel functions, are called system calls. This is special names to these particular functions that the operating system kernel provides to applications to be able to do things that it wants . And some examples of system calls are calls to allow you to start a new program or to read or write to a file on the disk or to exit .

And there more system calls that were going to look at as we study this course for. So, let us continue with this particular model where we said that the shell itself is written as an application and the operating system provides certain interfaces to allow the shell to start another program. So, what are these interfaces?

(Refer Slide Time: 28:48)



So, Unix provides a system call called fork and another system call called exec and these system calls are used to start a new program. Let us see how fork and exec are used. So, first let us talk about exec. So, let us say I am an operating system and an application is running on the top of the os. Let us say there is a shell running as application program; the shell can make a system call called exec.

So, exec takes an argument which says file name and what happens if a program calls exec system call is that; firstly, the operating system will search for that file name in the file system. So, it searches for the file name in the file system to find in the current directory; to find if there is a file with that file name. If it is there and it is executable, then it replaces the shell with that particular file name.

So, let us say the file name was browser, then the shell will get replaced with browser. So, exec is a way for one program to load another executable or from the disk . And it is done using the file name. So, there is a file name that the program gives that says I want to run this particular program, but the problem is that once you do that you yourself are no longer there.

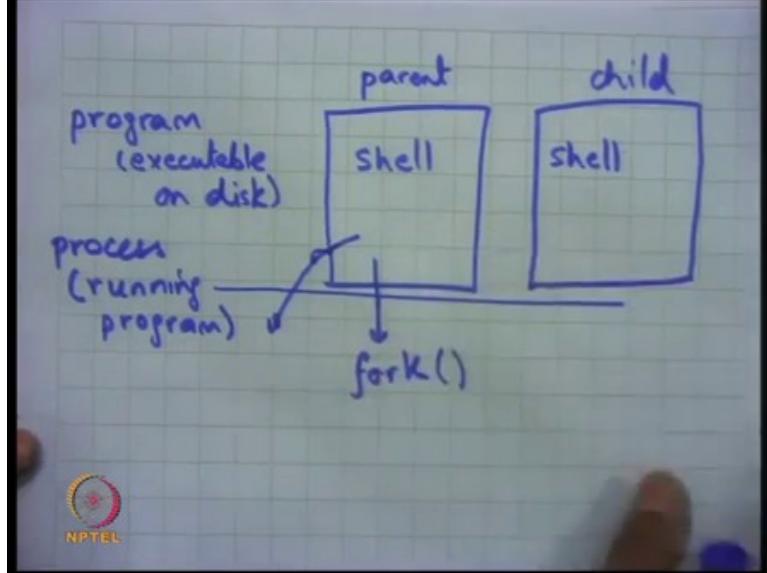
So, at the point when you call the exec system call, you are the one who is occupying memory. As soon as you call exec, the operating system removes you from the memory and instead loads the contents of that executable into memory and transfers control to it. So, essentially that means that if a shell program ever calls exec then the shell program will never get to run again. It will be the new program that will run and when that program calls exit, it is no longer the case that the transfer, the control will get transferred back to the shell unlike in the original Unix .

Because in the original Unix, the shell was part of the operating system kernel and so, it was possible to jump back to it, but here because the shell was an application and the application has called exec. So, it has completely removed itself from the picture . And all it is state has been wiped out and so, there is no way that operating system can jump back to the old program.

Notice that the operating system does not even know that this particular program is the special program called shell. It is just one of the different programs and it does not know what point it was in when the exec system call was called. So, it has no way of reconstructing that program back again. So, exec is one way to load another program, but it also means that I completely get washed out. And there is no way that I can return control to myself later on . So, that is not good enough to for us to be implement to implement the functionality of shell as we know it . Because the shell as we know it basically allows us to type a command, the command gets executed and when the

command exits; the shell can run back again. So, that is something that we want and exec is not going to do the last part which is return back to the shell. So, so the other system called fork can perhaps help. So, let us see what fork is going .

(Refer Slide Time: 32:44)



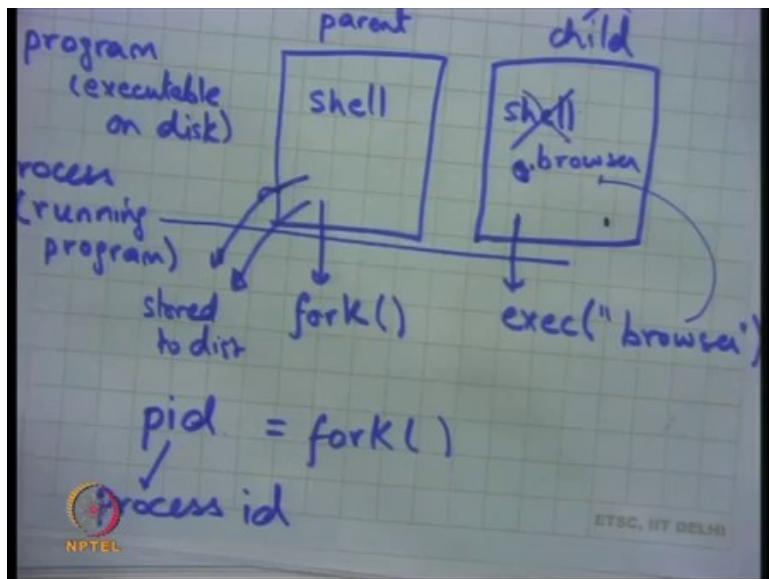
So, what is fork? Well, let us say there is a program called shell that is running. It calls a system called fork; what happens is when it calls the system call called fork, two different apps get created simultaneously identical to each other. It is like a parent.

So, one child one process creates an identical replica of itself using fork. So, they are two shells that get created and at any time only one shell can run. So, the fork system call is going to do is it is going to create two copies of itself with identical state and identical value of the program counter. So, they are going to both be executing the same next instruction when they call fork.

Of course, in our model we are saying that only one program can run it any anytime. So, what will happen is one of them let us say the; so firstly, one of them is called the parent; the one who called fork and the other one is called the child . The program that called fork is called the parent and the program that just got created new or the process. So, that just what created is called the child. So, a little bit of terminology here.

There is a program which exists as an executable on disk and there is a process which is a running program. The exec system call takes a program and converts it into a process; a running program. The process runs and it changes its own state. So, process has its own state in memory and it as a run it changes its own state and at some point it calls exit let us say. So, initially the shell was running at the process. The shell made the fork system call and it created two processes; the parent process and the child process . Now, at any time only one process can be active. Let us say and if it is a uni processing system.

(Refer Slide Time: 35:47)



Then, let us say the child process gets stored to disk and the parent process gets stored to disk and the child process gets to run. And so, that is the semantics of fork, but our real goal was to be able to implement the shell functionality. So, what can be done is that the parent process creates a child process and the child process now calls exec . So, what happens is the parent process still exists exactly at the point where it had forked the new process. The child process can now load the new program which basically means the shell gets wiped out and instead gets replaced by the browser .

The browser can get to run and at some point, the browser will call exit in which case the whole child process will get wiped out. So, we have seen that there is a system call called fork that allows you to create a new process, which is a replica of the process that called fork and there is a system call called exit, that allows you to stop the current process or completely free the current process and there is a system call called exec which allows you to replace the contents of the current process with the new program. Now using these two system calls; fork and exec, it is now possible to implement a shell like program as an application.

So, for example, a shell will fork first call fork, so it will create two copies of the shell. The child copy of the shell will call exec on the command that you gave, on the command prompt. The command will get to run, when the command finishes it is going to call exit. So, that process gets wiped out. The old shell, the parent shell, that was stored on disk gets loaded into memory and continues from where it left off .

So, that is , that is one way that you can implement the shell functionality. So, as opposed to having the shell as part of the operating system kernel, it is a much more modular way to actually have the shell also as a separate program; much more powerful way instead in fact, to have the shell as another application and have these special system calls called fork and exec that allow you to implement the same functionality as the shell.

In fact, the system calls fork and exec can be used by any application . The shell is just one of the applications and the kernel does not even know that there is a shell . It can be just any other application. So, any application when it calls fork and creates replica processes of the parent process and any process that calls exec is going to load the file into the current process .

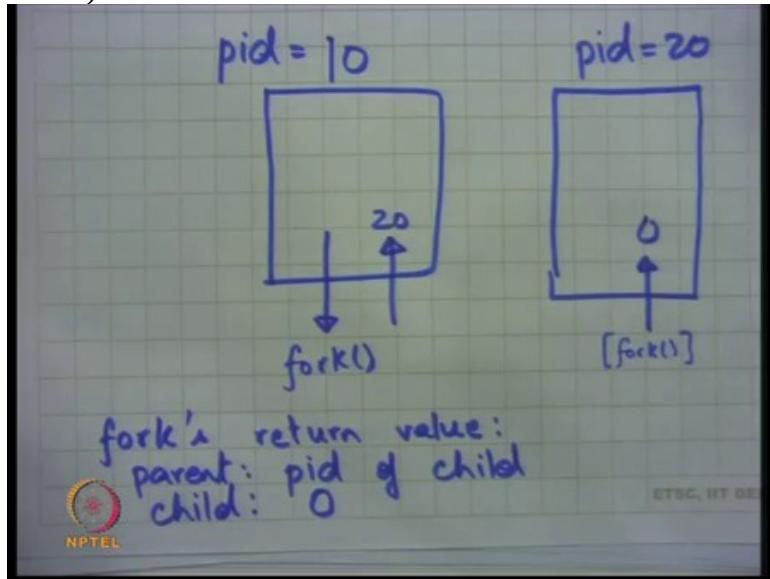
There are few things that I have skimmed over; one is when a system call, when a fork system call is called then there are two processes that return from the fork system calls. So, fork is a

system call that is called by one process, but returns in two processes. So, one process calls fork and two processes returned from folk at exactly the same program counter .

But there was a difference; I said the child will execute the exec system call next, if it wanted to implement the shell functionality; while the parent will just print the next command prompt. So, how do you do this? Because the two programs are completely identical, they are twins; how do you decide whether I am the child and I should call exec or whether I am the parent and I should actually display the command prompt?

Well, so, the fork system call has a return value. So, the syntax of the fork return call is; fork returns a return value which I call the pid or process id and the return values are different in the parent and the child. So, let me just draw this as a picture.

(Refer Slide Time: 39:59)

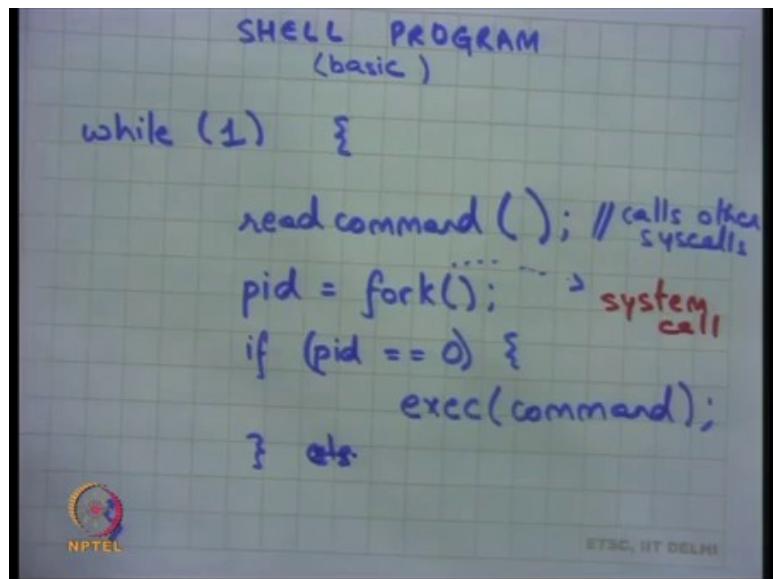


So, let us say there is a process. So firstly, every process has an id. So, let us say this is process 10 which basically means each process has an id. So, this process has an id of 10. So, pid is equal to 10. This process makes a system call called fork and the operating system returns from this system call, but it returns in two places.

So, it creates another process, let us say it creates another child process and this child has a pid of 20. Pids are assigned by the operating system. So, it can choose any way of assigning opponents thing and this one also returns behaves as if it has just returned from fork . This, particular process did not ever call fork, but it behaved as though it has just returned from fork. Both of them are going to return at exactly the same program counter except that the return value in the parent will be different from the return value in the child.,

So, in the parent the return value is the pid of the child. So, the return value in this case will be 20. In the parent process, the return value is pid of child and in the child process it is some number which cannot be a pid, let us say 0. So, here are the return value is 0. So, all that you need to do, all other all that the programmer needs to do is to check the return value. If the return value is 0, then I will call exec; if the return value is non-zero, then I will print the next command prompt for example .

(Refer Slide Time: 42:09)



So, let me just write this more formally. I am going to write the shell program, very basic it looks something like that shown above. It is an infinite loop(`while(1)`) just continue still the user types `exec` or just some something else to basically stop it, but otherwise it just keeps running. And it has some function to read command, how it reads the command let us just abstract it out. Let us say there is some function which says `read command`. It is going to make some system calls internally to basically read the command. This `read command` is not a system call; it is I am just using this function as a placeholder to say it will call other sys calls that we will discuss later.

And, it will say `pid` is equal to `fork`. This is a system call. Let me write it in another color system call . And then I am going to check, if `pid` is equal to 0 then `exec command`, whatever you read from here. Else, let us say else you just read the next command. So, what is happening; here is the program that will at a very basic level implement the shell functionality. It reads the command, calls `fork`. In the child process it calls `exec` and in the parent process it just reads the next command . So, this will exactly do what we wanted it to do basically; which means it will execute the new program and allow the previous program to do this to repeat the same thing again.

But the previous program will probably get to run only after when the new program has finished for example . So, this is sort of uni processing environment where only one process gets to run and let us assume that the child process gets to run first the child process is going to run and when it calls `exit` then it is going to go away and the parent process is going to get to run now.

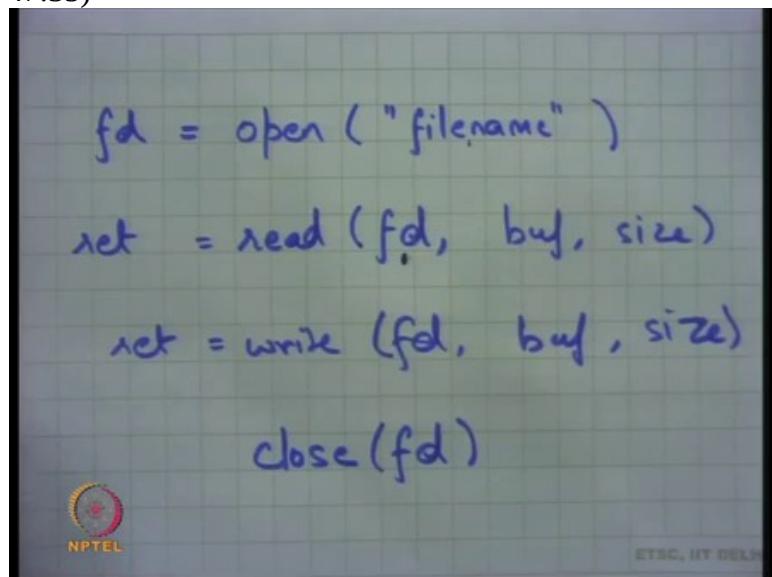
And the parent process is going to go back in the loop and read the next command . So, that is what is happening in this loop. So, there are many details or many things that I have omitted here, but it is important that before we go there you understand the functionality of `fork` and `exec` and the use of `fork` and `exec`.

Now, let us understand how do you read the command . So, typically we think of `read command` as something which is being read from the keyboard . So, the user is typing some command and he presses enter that is a command. So, how do you allow an application to read the key is being pressed on the keyboard . Similarly, how do you allow an application to be able to write characters on the console. So, both these things are mediated by the operating system and in general any resource that is a shared resource which needs to be shared across multiple applications is usually mediated by the operating system kernel .

Because if anything is shared then the operating system needs to make sure that there are no concurrent accesses by multiple applications; there different applications are not stepping on each other stores for example. So, operating system need to mediate in the middle. Some examples are, we just saw that the file system is a way to for the operating system to allow mediation of the disk blocks .

So, the file system is basically in on disk data structure; that is shared by multiple processes. And the operating system basically exposes functionality like open, read, write to allow applications to access this these disk blocks which are shared. Similarly, the keyboard and the console and that the other hardware devices are shared resources and different processes may need to access these shared resources and once again all these hardware devices, the access to these hardware devices is mediated by the operating system. One contribution of the Unix system was the general interface to be able to use across a variety of devices and other shared resources.

(Refer Slide Time: 47:35)



And this interface is open, read, write and close . So, I can say open a source which and I named the resource using a name . And it will let us say return a file descriptor. So, we have seen this before, but we are seeing this in the context of files only, but the same thing can be applied to other things like devices, like the keyboard and the console and how we are going see very soon.

So, when an application says open, that is the time when the operating system is going to check whether let us say somebody else is using that resource. If so, perhaps operating system; you want to say no I will not allow you to open it in which case the open can return an error .

So, typically these system calls return positive values and if they return a negative value like minus one; it indicates that the system called failed. So, if an application calls an open system call and it gets the minus one return value, it basically means that the device was not the file or the device was not opened. Once you open, if let us say you opened a file name and a file descriptor was returned which was a non-negative value, then that file descriptor can be used to as an argument to future calls which are read.

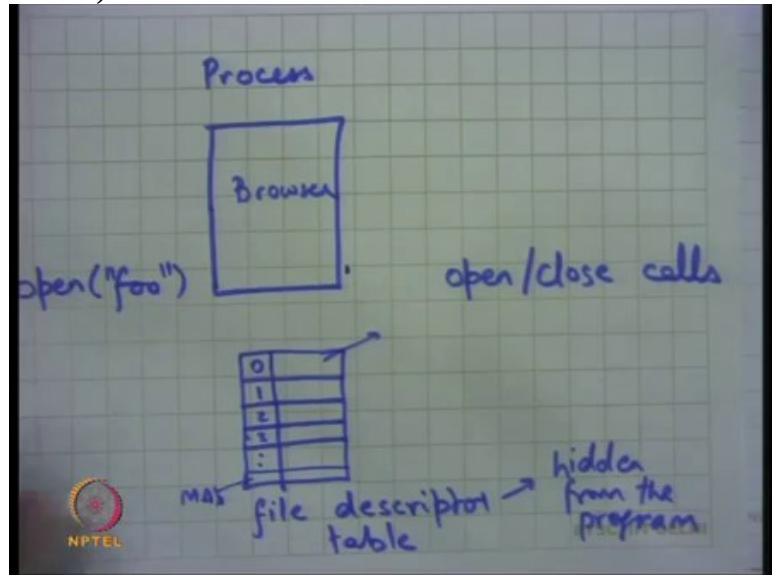
we have seen this before read buf size, read and write to that resource. Once again that the operating system can decide whether it allows read on this particular file descriptor or not. So, for example, if I try to read from the console, I should get an error. So, it has a return value which can be which can

be negative which basically means over that you are read was illegal. So, I should not be allowed to read from something which is an output device or output file.

Similarly, I should not be able to write to let us the keyboard or to a file which is a read only file and so on. So, the nice thing is because of this interface, the operating system gets to also in check or do perform access control for the applications to the shared resources. And then finally, you could do closed file descriptor to close that particular file; which basically means that I am done with this particular file and other people can let us say use it etc.

So, it allows the operating system to know who has which files open . So, let me take this a little further. So, how is the file descriptor returned? It turns out there is also a way, there is also a convention in which these file descriptor numbers are returned. So, whenever the file, a process calls open a certain file descriptor number will get returned and let us understand what this convention is.

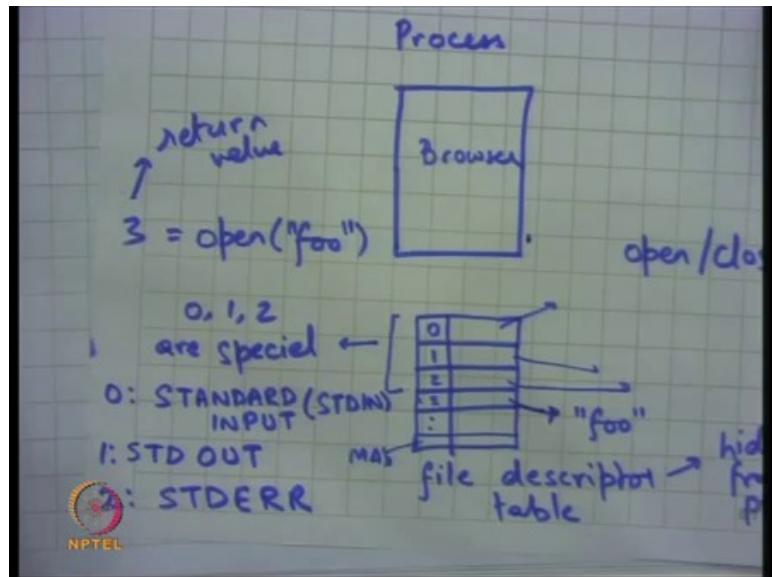
(Refer Slide Time: 50:52)



So, we said there is a notion of a process . A process is a running program. So, let us say I am drawing a process as a strict angle and there is a program that is running inside it, let us say the program is browser . Each process also has certain state called the file descriptor table. And, this file descriptor table is hidden from the program. I mean the program cannot access it directly, but the program can manipulate it using open and close calls .

So, when a program calls open; the operating system searches for the first available file descriptor and so, let us say I say open foo. I search starting from the beginning. So, the file descriptor table has key value pairs, the keys are numbers starting from 0 till some maximum value. And the values are structures which contain whether it is opened or not, and if it is open what are its weight specifications.

(Refer Slide Time: 52:33)

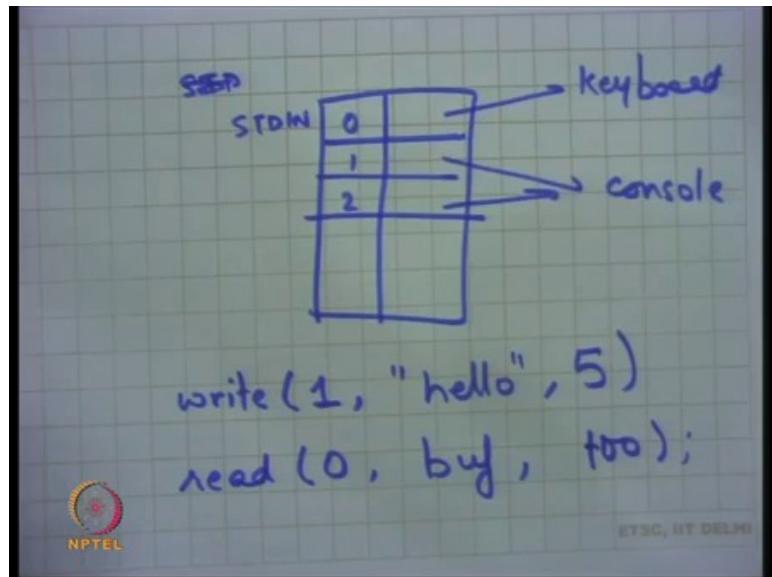


For example, 0 could be pointing somewhere, 1 could be pointing somewhere else, 2 could be pointing yet somewhere else and 3 may be empty. So, when I say open foo; what will happen is the operating system will go through the file descriptor table, look for the first unused fd which is 3 and it is going to put foo here and return 3 .

That will be return value to the application. The application can then use read, write, close on this file descriptor to read or write to this file as we saw them. The first three file descriptors 0, 1 and 2 are special. They are called standard input or STDIN, standard output that is 0. This 1 is standard output STD OUT and 2 is standard error. So, process has three special file descriptors 0, 1 and 2 which are, which referred to it is standard input or their special names for these three file descriptors and the names are standard input, standard output and standard error. The idea is that a program typically will read from its standard input and write to its standard output and if there is an error, it will write to the standard error output .

So, if I am a shell program then I will always read from my standard input and write to my standard output. I do not care whether the standard input is the keyboard or something else. I do not even care whether the standard output is the console or something else. The program is written, the semantics of the program that it reads from the standard input and writes to the standard output. And, somebody else let us say the operating system is responsible for deciding what is the standard input for this program and what is the standard output for this program . So, let us assume that the standard input.

(Refer Slide Time: 55:14)



So, let us assume that the standard input. So, zero or standard input is points to the keyboard and standard output points to the console and standard error also points to the console . So, now, if the program calls write(1,"hello",5) i.e, write on file descriptor 1; basically, says I want to print this on buffer which has five characters in it, then that program is effectively writing the string hello on the console . If the program says read from standard input, some characters into a character array buf which he which he may have declared of some size let us say 100, then it will read a maximum of 100 characters, but if the user pays 100 at enter before the 100 characters that is also fine.

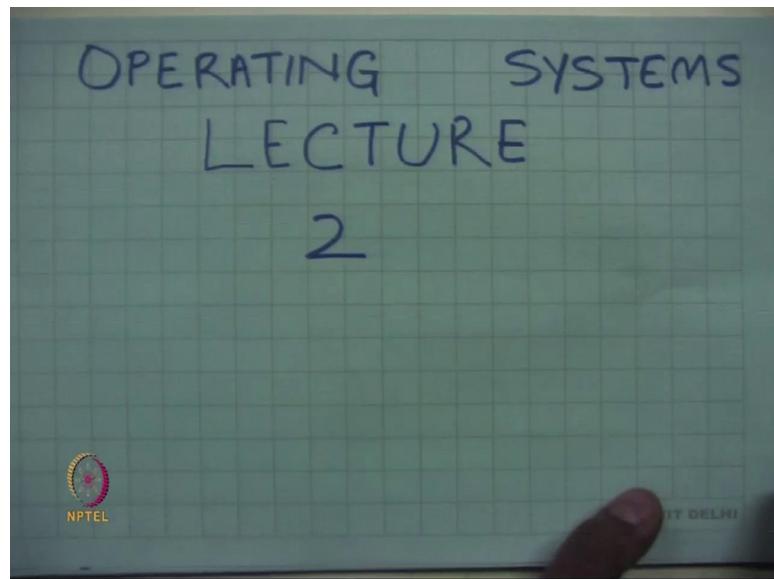
So, it is going to read some a set of characters from the keyboard from to this buffer . So, that is the way you read or write to or from the console or the keyboard . And, notice that very very elegantly and very smartly the operating system designer has used the open read write closed system calls to also be able to read or write to the devices and not just to the files. So, we are going to continue this discussion in next lecture and yeah that is it for today.

Thanks.

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

Lecture - 02
Introduction to UNIX System Calls Part – 2

(Refer Slide Time: 00:25)



Welcome to Operating Systems lecture 2. Yesterday, we discussed that software for computer system is usually very complex, it involves lots of different things including event handling for devices, running different programs, loading different programs, writing new programs and so on. And its important from a manageability and point of view to structure your system in such a way that it is; it becomes more manageable and that is the; that is in some sense the area of operating systems.

And we discussed then an operating system is in some sense the lowest layer of software that sits on top of hardware and it exposes certain API or interfaces to the applications so that they can use those interfaces to do all the things that they would want to do. And we said that in general designing such an API is not revealed; its little complex and we started with looking at one particular operating system Unix which was one of the first very successful multi processing operating systems and on which many current operating systems are model and we started looking at what its interfaces .

(Refer Slide Time: 01:47)

```

while (1) {
    write (1, "$ ", 2);
    if (exitcode) readcommand(0, Command, args);
    if ((pid=fork())==0) {
        exec(command, args);
    } else if (pid>0) {
        wait(0);
    } else {
        printf ("Error in fork");
    }
}

```

In particular, we started looking at one powerful program called shell which lot of us use in daily life and see how a shell is implemented on top of Unix substractions . And so we said look the shell is actually implemented as this loop which is denoted by the while statement. In this loop, the first thing it does is write to its file descriptor number 1, a string of length 2 that is the dollar form that you see on your screen.

The file descriptor 1 is a special file descriptor which points to the standard output, the standard output could have been opened by the shell itself for example, or it could have been inherited from its parent or whatever other ways this the file descriptor could have been initialized. Moreover this file descriptor could point to a file, it could point to a device, it could point to the network so that allows you to write really environment agnostic code when you are coding something like that. So, the same shell can execute both on a single code and can execute on your screen.

Then there is this is imaginary functions that I have written here it is called read command you can; so what this read command function does is, it invokes the reads system call . Once again as a reminder a system call is the function that a kernel provides to an application . So, at inside the read command function it must be calling the read system call and it must be it and its calling the read system call on file descriptor 0 .

The file descriptor 0 is also a special file descriptor which refers to the standard input of the program and once again, it can be initialized in any way exactly as it was initialized for the standard output. It reads commands from the standard input, parses them into the

strings called command and args. You can imagine that this command is in an array of characters and an arguments is also an array of strings where each string is an array of characters.

So, once it has parsed the command which it has read from the standard input of the shell, it needs to actually execute that command . So, let us assume that the name of the command can be parsed as a name of file on the file system and so what you are going to do is you are going to execute that file, you will going to pick up that file from the file system and you are going to execute it.

But once again you cannot just execute a file in the context of your own process. The shell is running in a certain container which we call the process which is again an abstraction that Unix provides and so the way it works is that Unix allows you a system called fork, which clones the current process. So, which means it makes a copy of itself and creates another process which we call the child process. The new copy of the process is identical to the parent process except that its process id is different and the return value of the fork system call is different .

So, we assign the return value of the fork system called to this variable called pid which is one local variable and then we check its return value if that return value was 0; it means that we are currently executing this code in the context of the child process; in which case we execute another system called exec which takes the command and the arguments and tells the OS to execute this command in the context of the child process .

So, the parent process remains completely unaffected by this operation of exec which executes only in the child process. The semantics of the exec system call are that it will completely forget about the current process in the terms of the memory contents; load the executable into the context of the current process and started running .

So, in other words; once you call the system call exec, control actually never reaches the point indicated in the above picture(just after exec system call). Because once you have called the exec system call you have loaded the new executable, you have transferred control to the first instruction of that executable and that executable has completely forgotten the context in which it was loaded . So, the process is completely sort of reinitialized in some sense.

So, you could write anything there, but it will not get executed. That is the semantics of exec never returns, but an executing process or any process can exit . So, there is another system called exit which , this is again a system call or a function provided by the kernel; that a program that a process can call and the semantics of the exec system call are that the process will terminate and all the resources belonging to that process will get read .

So, all memory etcetera will get read, whatever belongs to that one. So, that is the semantics of the exit system call. Often, when a parent spawns or forks a child process; it was also interested and what happened inside the child process whether the execution of the child process exceeded, whether it failed or what was its return value for example .

So, Unix allows you to specify an argument to the exit system call which let us call the exit code or you can which is just an integer which can be thought of as a return value of the process as it exits ok. And the parent; so that is that is how the child executes, it calls the exec system call and the exec system the program that is loaded inside the exit call system call may or may not call exit at some later point of time.

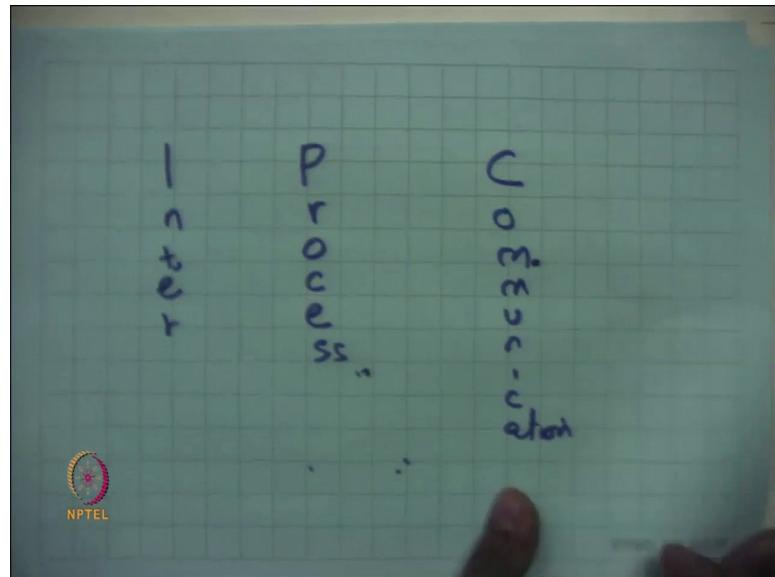
The parent can monitor what is happening to the child by calling the wait system call . So, the semantics of the wait system call are; in this case I am calling it with the argument 0, it basically means wait till any of my children exists . So, as soon as that child; so in this case a parent will only have one child process at most living in the system . So, the shell process is one program; one process and you will only fork and then you will wait.

So, at most you will have one outstanding child in the system and so when you calls wait 0; you basically waiting for that particular child that you just want. So, the semantics of wait is that will block till the child is running and it will return as soon as the child exits. Further, the return value of wait is the exit code that was passed as an argument to the exit system call in the child .

So, if the child called exit with return value 10; then the return value of the wait system call in the parent will return 10. So, that is the limited way of inter process communication between the child and the parent , it is a question. So, what happens if the child does not call the exit system call, but crashes for example, it touches illegal memory and call the segmentation fork . Even in that case the wait system call returns with the appropriate error code, indicating what happened to the process.

So, either either you will get the exact code in inside the return value or you will get us get a special code which indicates that the process crashed and why did it crash. The parent can get that kind of information for a child; it is a limited form of what are called inter process communication.

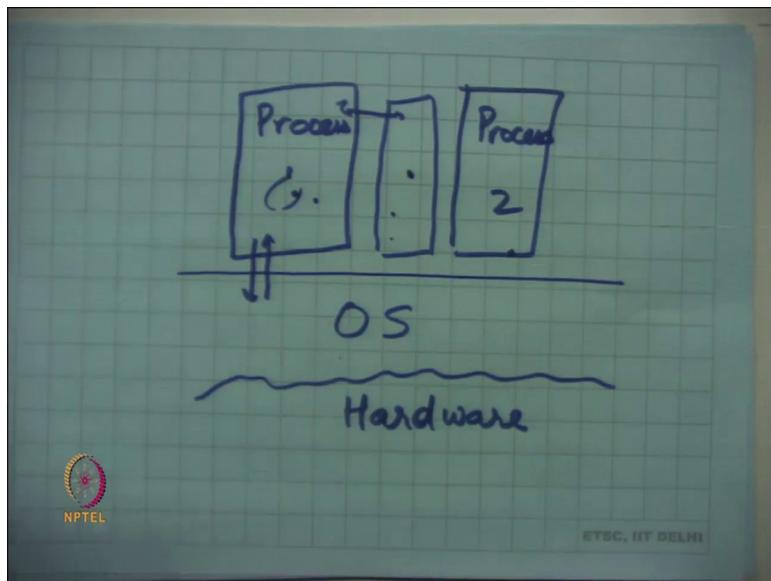
(Refer Slide Time: 10:07)



I P C; Inter Process Communication.

Once again, if I have to draw the diagram of the OS, there is the OS, there is let us say the hardware and there are different processes process 1, process 2 and so on . These processes are making system calls to the OS and getting answers to the system call.

(Refer Slide Time: 10:21)



One of the system calls is the fork system call which actually creates another processes such that it is identical to the original process. These processes are more or less isolated from each other. So, if I am if I instantiate a variable in some process and I write something to it, it is completely in my private space as opposed to a variable that is instantiated in other process.

However, often you need to communicate between these processes for various reasons and one way to communicate is what we saw is the exit code between the exiting process and the waiting process. This form of inter process communication is very limited because it is just only be used between a parent and a child, we cannot use it for any two arbitrary processes; also it can only be used if the child exits. So, it is a limited form of inter process communication and it only is useful when one of the processes are actually exited .

We are going to see more types of inter process communication very soon. Anyways, so exit code is going to tell you what happened to the child process. And most shells would typically have an option which say whether you want to display the exit code with which you can find actually whether your command succeeded or failed.

One of the standard conventions on Unix for example, is that if a process execution succeeded; then it will exit with exit code 0. If it failed then it will exit with exit code 1 or it can or some nonzero exit code indicating the failure condition. And the third if condition in the above picture is what; so if the return value of fork was 0 which means it is a child.

The second one was if its greater than 0 which means it was parent and the other the remaining one is when its less than 0; which indicates error in the fork system calls; the fork actually never happened . If the return value of fork is less than 0, it basically means that the fork never happened for whatever reason ; it can be many reasons for example, the process ran out of memory. So, this kind of interface though seems rather easy now was actually not so trivial to come up with when it was first in such .

So, there were many iterations before of what this interface should be before Unix was actually became popular. And as you can see this interface of; so this interface really involves something like a fork system call and exec system call, the ability to name file descriptors and to be able to call open, read , close on these file descriptors. The exit system call which returns an exit code and the wait system call which allows you to wait on an exit code and all these things what they allow you is composability.

So, it allows you to compose one program inside another program, able to spawn another program and treated just like a function call for example. So, you just one another program let it run; let it run asynchronously, collect its return value; it sort of becomes an asynchronous function call. You can also pass arguments to that one process.

So, specific arguments with that spawn process by saying what the file descriptors are going to be initialize the standard input, standard output, standard error; error based on what you want it to be and that sort of gives you a very nice composability, composable setup and allows you to treat programs as tools which can be composed and fitted in any place in an environment that you like to. So, in that sense it was; it makes things very clean and the and useful and in fact most operating systems today some flavor of this kind of an interface.

We also saw how if I wanted to redirect certain file descriptors to other resources than what shell is currently using and I could do that by inserting appropriate code before the exec system call inside the child container . So, for example, when I fork something, the

semantics of fork are that the entire file descriptor table of the parent gets copied into the child .

But, if I wanted to say; I want to redirect the output of my child process to some file let us say, then what I will do is I will close the existing standard output and then I will open the file and the once again the semantics are then open, will return the first available file descriptor in the file descriptor table which means standard output will get reinitialized to file.

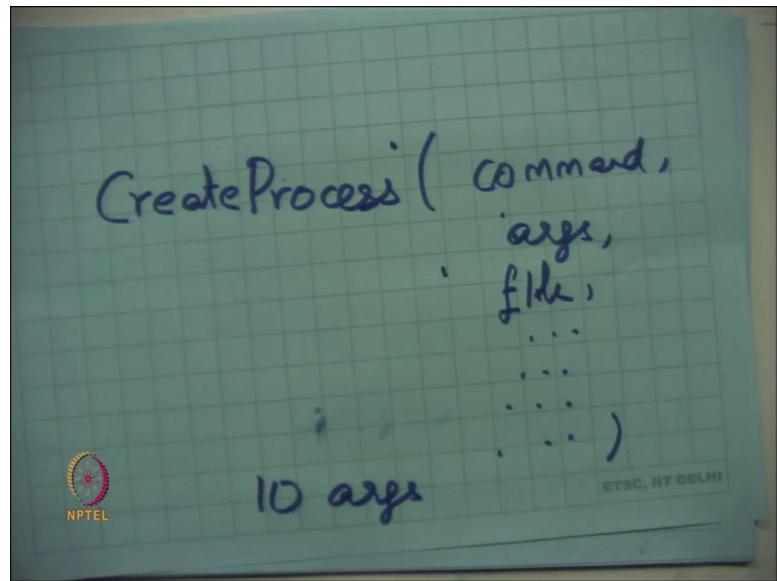
And now exec of ls can take place notice that in this I dint have to change the code of ls at all; ls was its composable in that sense because I just changed the environment and then invoke ls with that new environment and now ls behaves as exactly as I wanted it to be; This is one interesting example of composability.

Notice that this is possible this; this way of redirection is made very easy because I have separated this process creation into two system calls. One is the fork system call and the second is the exec system call and now I can do some things in the middle between the fork and the exec system. It is not very clear why this is a very very interesting choice really. Because you may argue well I am it is a little peaceful to do things in this way.

Because I first create a copy of myself using the fork system call and then overwrite the copy with something else . It seems wasteful that I first copy the entire contents and then completely overwrite it with something else. So, the first copy was really not needed and why am I doing it; I am doing it because Unix allows that is the only way Unix allows me to create a new process. So, there is the seems; it is like it is a performance problem.

However as I have discussed so far it gives you a lot of flexibility. Because you can do things between fork and exec. In fact, Microsoft windows does not do things in this way ; so, it does not do fork and exec.

(Refer Slide Time: 17:27)



It has a system call called create process. So, I guess the Microsoft windows engineers; OS engineers felt that the fork system call is too costly and we do not really need to do this and so they have a system call which is create process which takes; let us say a command and arguments and the semantics of the system call are that it will start a new process and start this command in that new process. So, there is no fork ; so there is no copy being done and so you avoiding that wasteful operation, so called wasteful operation on Unix .

However, as you can imagine; the new command may need different types of environment. For example, if I wanted to implement shell redirection on something like this; there is no there is no clear way of how to do it . So, if I wanted to say ls greater than file; who not quite possible unless I change the code of ls. So, what they did was they added more arguments here, like file descriptor file descriptors and so on environment variables and so there are roughly; 10 arguments in this system call.

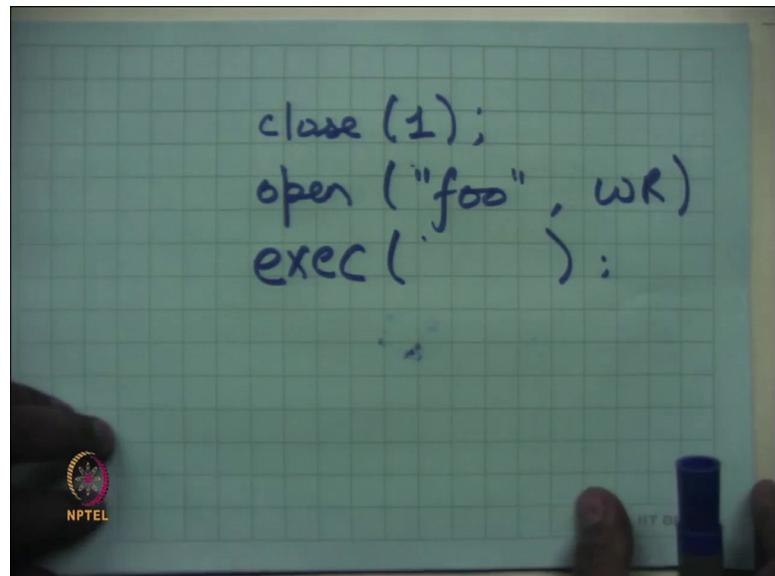
So, window system call says create a process with this command and these arguments and this environment . So, that makes your system call rather bulky, but arguably it perhaps more efficient than Unix ok. So, there is a; there is an interesting trade off I mean here is an example of a trade off between clean interfaces, small interfaces and performance interfaces.

It turns out that Unix in its fork and exec is not all that costly ; as we are going to discuss later in the course, it is possible to implement fork in a very fast way . Using what are

called copy on write optimizations where when you create a new process, you do not necessarily copy the entire process up front, but you just create pointers to the original process with read with and set the entire process read only in read only mode and only copy when something is written in the new process.

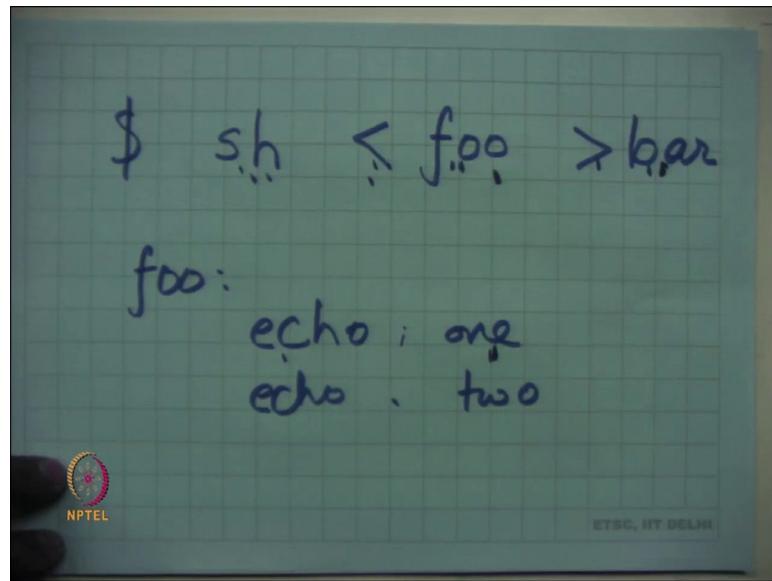
In the common case when the new process actually just exits to something else, your fork was pretty much free . So, interesting example of 2 D algorithm operating systems doing the same thing in different ways ok; last time we saw how shell implements redirection .

(Refer Slide Time: 20:12)



So, the way it implements redirection is that before it calls exec; it closes the standard output, close 1 and then let us say opens foo and let us say write mode and then calls the exec whatever command there is and so that gives you redirection of commands on the shell .

(Refer Slide Time: 20:43)

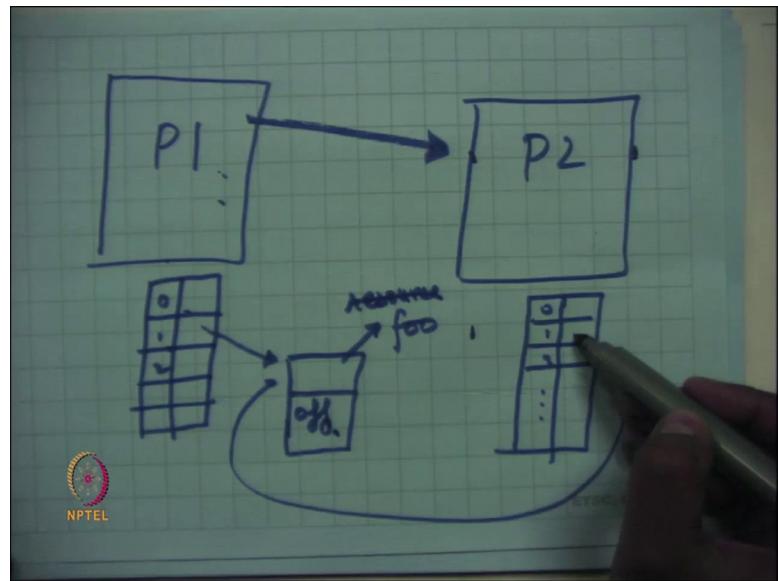


I could also do something like sh less than foo, greater than bar. What is this doing is; it is redirecting the standard input to foo and rewriting the standard output to bar and if you do it in pretty much the same way as we discussed earlier. You will close the standard input, open foo, close the standard output, open bar and execute sh .

If I wanted to execute a shell script; all I need to do is let us say this is my foo file; it will have let us say a few commands echo one and echo two. So, now foo can be a shell script and when I say shell is less than foo; the shell script gets read as input to my shell and that gives you shell script foo .

So, very composable here when I said echo one, I did not care about where the data is going I am basically saying I want to write it to standard output and the standard output was actually initialized much later and the invocation of the shell script foo . When I say shell sh is less than foo is greater than bar, it basically executes the shell script and prints one and two on the shell on the in the file bar.

(Refer Slide Time: 22:23)



So, let me just draw the file descriptor once again. So, let us say there is a process P 1, this is another process P 2; there is a file descriptor table that is associated with the process P 1 and there is a file descriptor table associated with process P2 . The process P1 can only access this file descriptor table using system calls. So, I cannot a process P1 cannot just access this table directly; this table is sort of hidden from the process and the only way it can actually manipulate or read these tables is using read, write, open, close system calls.

A single entry in this file descriptor table points to a structure which says where is my resource and then it says what is the offset of that resource. So, the offset is maintained inside this structure. So, let us say the resource is a file foo and then there is an offset inside the structure which says what is the current pointer at which file is writing. So, when I call a system call like write on file descriptor 1, it basically just starts writing at the current offset off.

If there is another file that wants to write its again going to write off and so it is going to off. So, off is going to get implemented on every write system call, that is how two successive writes give the feeling of appending to the file; it goes to off. So, that is also part of the semantics of the write system call. So, the offset gets incremented on every system call it read or write.

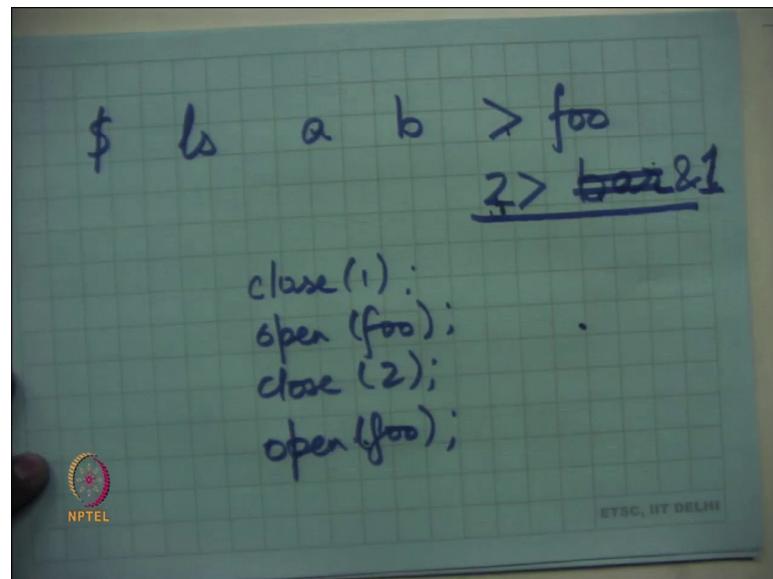
Now, let us say and once and one thing I wanted to point out is; so when a process P 1 forks a process let us say fork P 2; then the file descriptor table gets copied and the

pointers are shared . So, the same structures get shared and this allows you to do what is called file sharing . So, the child could be writing to the same file and the parent could be writing to the same file and their outputs would now get intermingle; in any arbitrary order depending on the execution order.

You could insert synchronization between your child and the process to decide the order in which you want to write to this file, but in any case this implements file sharing. There is another way of inter process communication ; So you a process forks another child process, the file descriptor table gets copied, the file gets shared whatever one process is writing is now visible to the other process for reading.

However, if process P2; let us say calls close on file descriptor 1; that has no effect on the file descriptor table of process P1 . Because the file descriptor tables themselves have been copied; it is just a pointer that have been shared here . So, if the process P 2 calls close; absolutely no effect on process P 1.

(Refer Slide Time: 26:04)

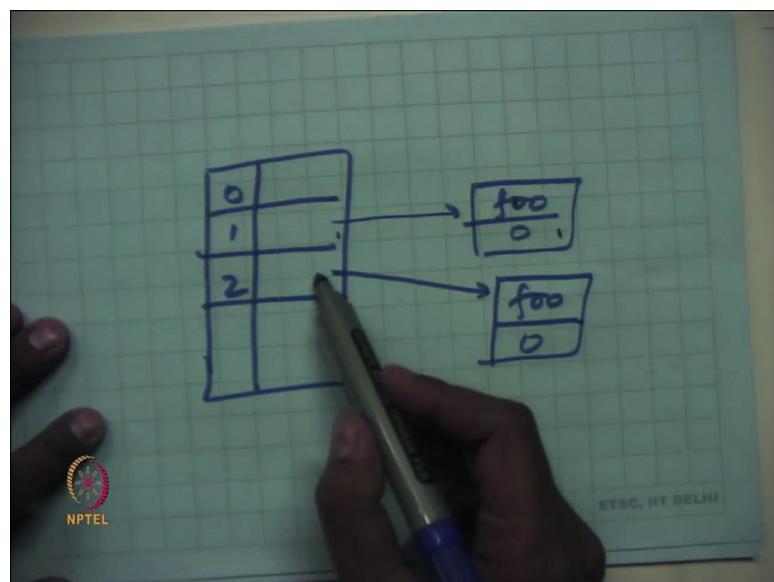


The shell also allows you to do something like “ls a b > foo” and so this means redirect the standard output to file foo and “2 > bar”; which means redirect the standard error code to file bar that is it very easy and you can imagine how this can be implemented. But what it also allows you to do is it allows you to redirect standard error code to the standard output port.

So, basically saying redirect the standard error; whatever is written to the standard error to the same resource that is being pointed to by standard output. So, that is the syntax for that one shell is 2 greater than and 1 depends on different shells, but let us say this is syntax for redirecting standard output standard error .

So, how would one do this? How would a shell implement this? Well one naive way to implement this is to do the same thing which is close(1), open(foo), close(2), open(foo) ; things like the correct way of doing things ; let us see what happens.

(Refer Slide Time: 27:36)

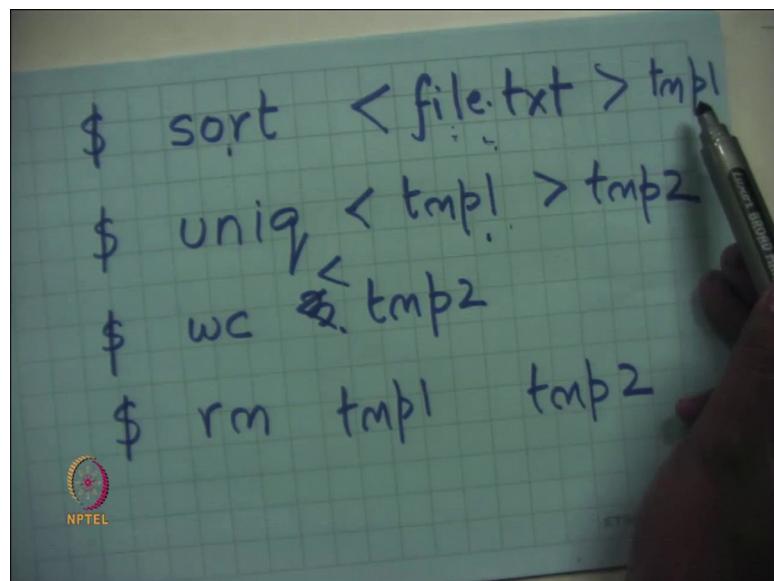


Here is a file descriptor table. So, when you call close(1) and open(foo); this point this gets initialized to foo and offset 0 and when you call open and open(foo); this gets initialized to foo and 0 . And now when the program was on standard output; it is going to write to file foo at a certain offset and then call write something to standard error, it is going to write foo and now what is going to happen is that they are going to overwrite each other .

What you wanted in this command was that they should get appended ; not overwritten . So, what do you need? It would have been nice in this picture instead of having something like this; I have something like this. So, they have a shared offset and each time they write either to standard output or standard error; the same offset gets incremented . The current set of system calls that we have discussed so far do not allow this kind of manipulation or the file table descriptor.

So, Unix has another system called; called dup ; it basically means that duplicate the pointer in the file descriptor table . So, the way I am going to implement this is I am going to say close(1), open(foo), close(2); instead of saying open(foo) want to call dup(1). So, that is going to have the desired effect of having a shared offset between standard output and standard input. Now, let us look at more interesting unique instructions.

(Refer Slide Time: 29:50)



So, let us say I wanted to; I have a file called file dot text and I and I have a utility which takes the file and sorts it . Let us say the it is an this is utility this is a program called sort just like there is a program called shell and there is a program called ls; this is a program called sort which takes input from its standard input and sorts it and prints the output to the standard output .

Let us say what I wanted to do was I want to first sort all the five all the strings in this file called file.txt. Then there is a program called unique which takes a sorted file and eliminates all duplicate entries from that; so just retains the unique entries in the file. So, one way to do this is I assort file dot text and redirect it to let us say some file temp 1, then call unique on temp 1 and redirect it to temp 2 . Then let us say I wanted to count the number of unique words in this file.

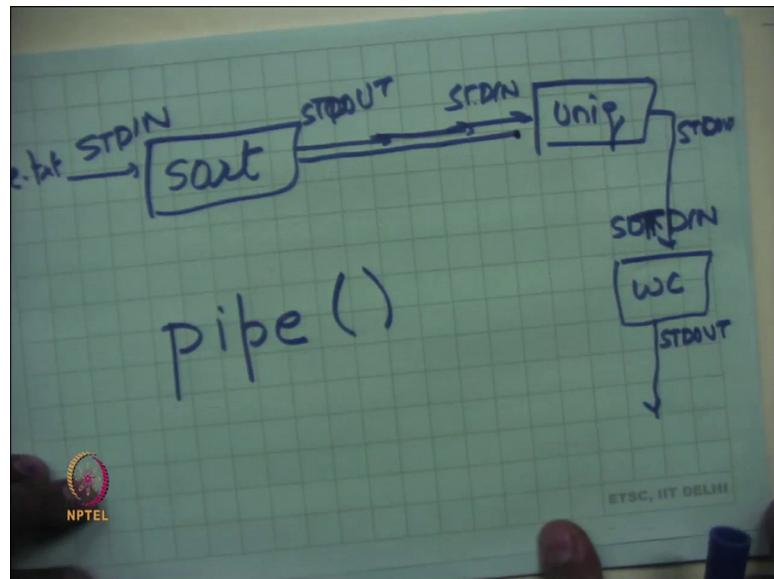
So, there is another command called wc with word count and I want to say a word count less than temp 2 or let us say I just say work let us say work count less than temp 2 and

so this the combination of all these three commands is going to count the number of unique words in this file called file.txt. Sort the file find the unique elements and count the number of unique. Of course, I have created two extra files in this the process; so maybe the last thing I need to do is remove temp 1 and temp 2 .

This kind of a thing is also very common where the output of one program needs to be fed as input to another program . This is once again very very consonant with the concept of tools where there are multiple tools and you want to say I want to use this tool and then feed the output to this tool and then this tool and so on . If I am doing it in this way; let us say my file is hundreds of gigabytes or terabytes large. I need a lot of temporary space to be able to maintain this information number 1.

Number 2, I am writing extra things to the disk and writes to the disks are generally costly and so I am making things very inefficient. So, I am first writing the whole thing to the disk, I am reading the entire thing from the disk then writing another set of data to this and so on and this is all very expensive.

(Refer Slide Time: 32:45)



Ideally, what should have happened is that I should have had a way on my operating system to specify that here is a program sort, connect its standard input to file dot text and connect its standard output to the standard input of unique. And then connect the standard output of unique to the standard input of wc and connect the standard output of wc to whatever the standard output of the shell .

Because you basically saying just say the shell was executing the console; you want the output on the console . So, if I was able to do this; if I was able to specify this then it would be very nice because I do not need these extra its temporary space and I do not need extra temporary time to create these space the space .

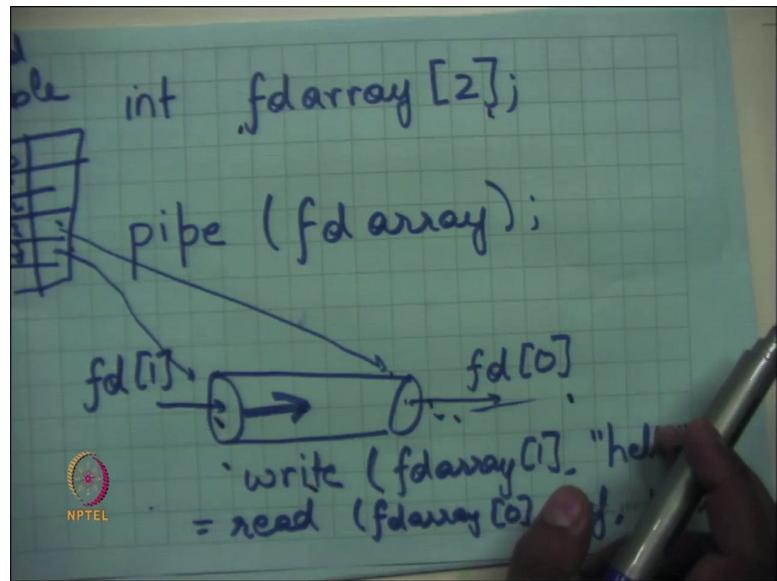
So, once again this requires manipulation of the file descriptor tables in certain way and once again I can know a process is not allowed to manipulate the file descriptor table in any way it likes. By the way why does not the process allowed to manipulate it file descriptor table in any way it likes?

For security reasons; you basically saying that these file descriptor tables are referring to shared resources; like the console, the file system, the devices and the OS needs to interpose on any request made to these resources. So, the OS will check on an open; whether this process is actually authorized to access this device.

For example, if this process is being run by the root user then it can access certain devices whereas, if it is not running as it root users cannot access certain devices. So, there is the ways I mean; so the OS needs to provide gates through which a process can enter; the process cannot just walk over my file descriptor table anyway it likes for securities. But now I need and then other OS is basically; the OS designer is basically choosing the write sort of system calls such that the processes are still able to do what they need to do must try.

So, I want to be able to do something like this and this; this connection between one file, the output file descriptor on one and the input file descriptor on another is facilitated by a system called pipe. Pipe was a very very new concept that Unix introduced and very successful since then ; so let us see what pipe does.

(Refer Slide Time: 35:55)



A pipe is a one way communication channel ; so here is an example shown in the above diagram. Let us say I create an array which is my fd array of size 2. So, the 2 fds in this array and I call pipe on fd array. So, this is the syntax of pipe as I am supposed to call the pipe system call on an array of integers of size 2 and what it is going to do is its going to create two file descriptors in my file descriptor table and put the output end of that file.

So, it is what is going to do is its going to create a pipe inside the OS. Once again this pipe is inside the OS and the process cannot just access it directly; it has to access it through gates which are system calls, it creates a pipe and it assigns two file descriptors to the two ends of the pipe .

So, for example it just let say there is the file descriptor table. what is going to do we just going to find the first two available file descriptors in this table; once again walking from the top. Assign the first one to the input end of this file and assign the second one to the output end of the file .

So, let us say the two file descriptors are available for 3 and 4. So, 3 is going to point to the input end of the file or to the depending on; so let us take this pipe as the entities. So, the output end of the file is the first one and the input end of the file is the second one .

So, with that in mind if this program says pipe fd array and then it calls on fd array[1], let say says hello string of length 5 and then it says read fd array[0] in some buffer buf and let us say its maximum length is whatever 100, then what I am going to get is answer 5 and the string hello written inside buf.

So, what has really happened is when you call write; it wrote 5 bytes on this file descriptor which means 5 bytes were stuck into this pipe. And when you called read on this file descriptor; these 5 bytes will read of the output and out of 5; what you have done is to recall that file descriptors point to resources. So, far we have considered resources which are either external externally facing devices like console or sealcoat or they have files in the system which are again; So, there is a file system let us say. Here a file descriptor is pointing to an internal resource which is a pipe; that is created using the pipe system call . And both ends of this resource are internal ; in the case of a console the output end was in the OS. So, the input end was in the OS and output end was on the screen, but here the input end is also in the OS and the output end is also in the OS; that is one way of think about it .

So, when I write to the input end of this pipe in fd array 1 and then read from the output into this file I get what I wanted to I get what I wrote . So, this does not sound very interesting; I mean if I am going if I just write and then I read what is the point of doing this through a pipe, I could have just done it internally; I did not have to involve the OS in this doing then, but what this allows you to do is allows you to do inter process communication across processes.

So, as you can imagine if I call the pipe system call; I have initialized my file descriptor table in this way and then I call the fork system call, the file descriptor table gets copied to my child with the pointers remain the same; so they still share the same pipe. Now, the parent writes to the input end of the pipe and the child reads from the output end of the pipe; you have implemented inter process communication .

(Refer Slide Time: 41:00)

```

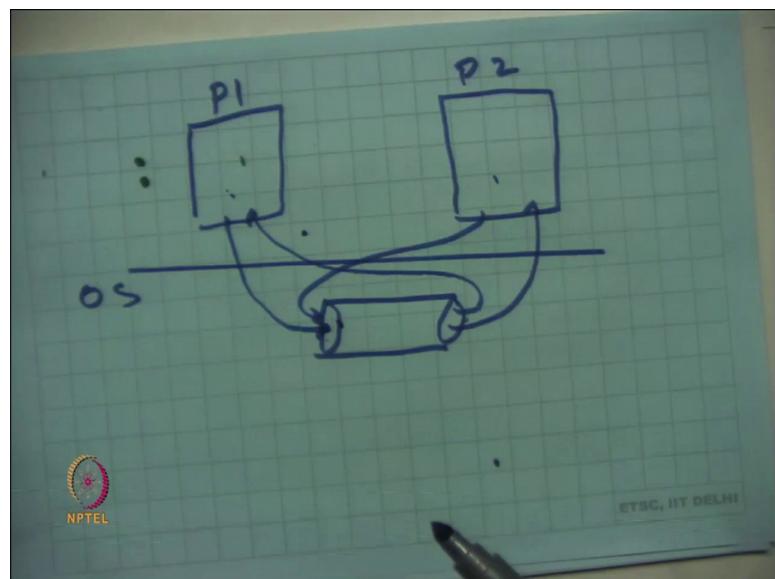
pipe (fdarray);
pid = fork();
if (pid > 0) {
    write (fdarray[1],
           "hello", 5);
} else {
    read (fdarray[0],
          buf, 100);
}

```

So, let us see how this works; I could call pipe fdarray, this is going to create a pipe . Then I am going to save fork(), this is going to duplicate the file descriptor table when; once a duplicate the file descriptor table the pointers get shared; so the pipe gets shared .

Then I can say if pid is greater than 0 which means I am the parent; I write to fdarray[1], hello . And else if I am the child, then read fdarray[0], buf and whatever 100 all in this case the parent writes to the pipe and the child reads from the pipe . I created a pipe, I forked after creating this pipe which means that the pipe now gets sheared.

(Refer Slide Time: 42:43)

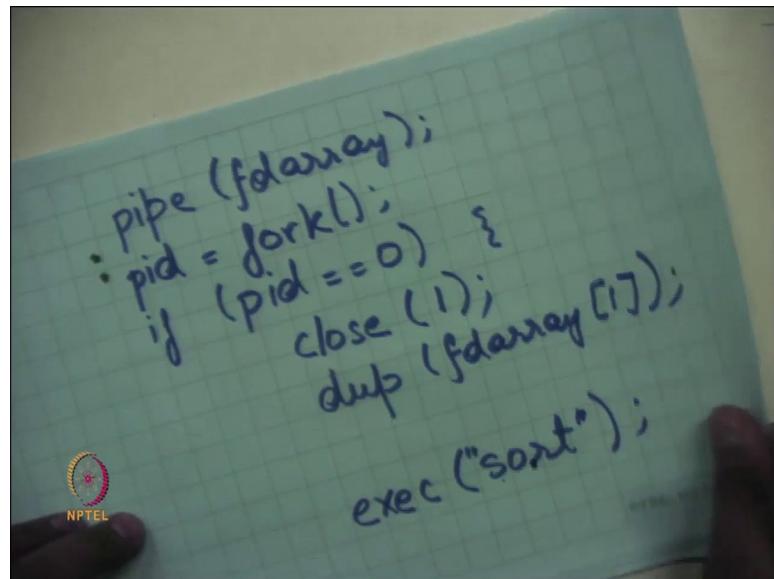


Let us just see what happens here is process P1, here is the OS, the process created a pipe. So, the pipe gets created inside the OS space which means it cannot be accessed directly by the process, but it can be accessed using read and write system calls.

Then the pipe has pointers which are represented by file descriptors. So, this is let us say fd array 0 and this is fdarray[1] and this is fdarray[0]. At this point the process P1 calls fork which means another process gets created P2, but the file descriptor table also gets copied which means that even this process has file descriptors which point to the same file; same pipe.

So, now if this process writes to the input end of this pipe either P1 or P2 can read from this pipe; they can call read and also. One way to use this is to have P1 to call write on the write here and P2 to call read here and that way you can pass information between P1 and P2. So, let us look at the example that we had earlier; I wanted to create pipes between sort, unique and wc programs and so I wanted a picture something which looks like this file dot text sort standard output connected to standard input of this and so on.

(Refer Slide Time: 44:52)



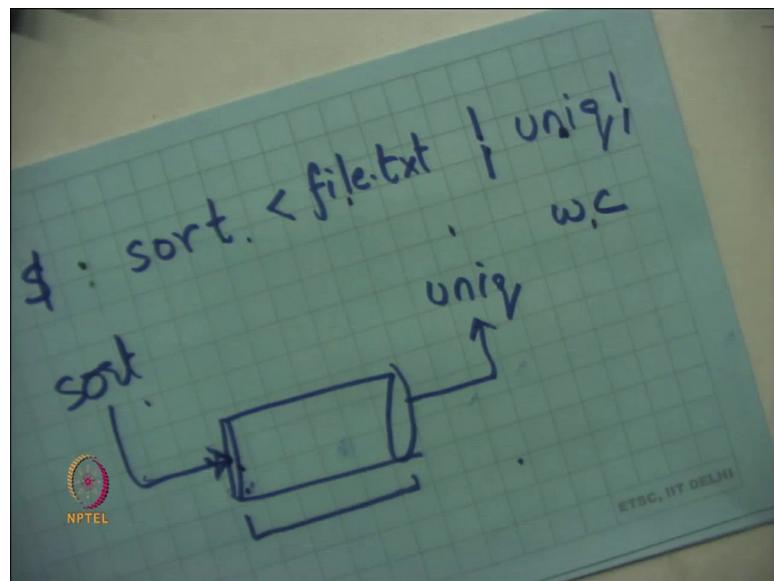
And so one way to do it is you call pipe on fd array, then you call fork and then you say if it is equal to 0 which means I am executing in the child context; then close the standard output, I want to connect the standard output of my child to the standard input of some other process. What I am going to do is I am going to close the current standard output of the child and I am going to call dup on fd array 1. What this is going to do is; it is

going to copy the file descriptor value in fdarray[1]; the point in fdarray[1]; two standard output.

So, now the standard output is connected to the input end of the pipe; fdarray[1] was input end of the pipe. So, when I did close 1 and the dup of fdarray[1]; I have connected the standard output of my current process to the input end of the pipe ; fdarray[1], . And then at some point I can say exec command the exec sort. I am hiding a lot of syntax just, let just understand (Refer Time: 46:40) when you execute sort; sort is going to write to its standard output and the standard output happens to be a pipe in this case .

Separately, I will connect the standard input of the other process that I am going to fork unique in and I am going to connect a standard input to the other end of the pipe and that way I am going to have a communication between sort and unique.

(Refer Slide Time: 47:04)



So, the syntax for this one shell is sort less than file dot text pipe and this is the pipe character on the keyboard I think it is and then say sort and then say unique and then again pipe and wc . This does exactly what we saw earlier which means its sort is going to take input from file dot text, pipe it to this pipe; pipe the standard output to unique. Unique is going to take the input from the pipe and pipe its standard output to wc; wc is going to take its input from the pipe and pipe this and print the standard output on the console.

And what I want you to think about after you go back is how we are going to, how the shell will implement some this syntax. So, this is the syntax that is provide the shell; its implemented by the shell using the file system call and I want you to write code to be able to do that . It is going to be something very similar to this, where you create a pipe, you fork and inside the child you manipulate the file descriptor so that it standard output now points to the pipe and then you exec the program that you wanted to .

So, when I created a pipe; there is the pipe sort is writing here and let us say unique is reading from here. what happens? So, now there is an interesting scheduling question that comes up; what happens if sort gets to execute a lot and unique has not even started . So, now sort is just producing a lot of data and unique has not gotten chance to run . So, now that brings to the question how fat is this pipe or how large is this pipe.

So, there are default values with the size of those pipes buns; so they and always there will be a producer to the in this pipe and there will be a consumer in this pipe. If the producers producing too much and the consumer has not even is not consuming at the same rate and eventually the pipe will get full; depending on the size parameter and the size is typically and if you tens of kbs; let us say.

So, once the producer gets full; the operating system can figure this out because the only way the program can produce is using the write system call and so the operating system sees the write system call it says where it is going its writing to this pipe it figures out of the pipe is full; it says let us block this process. When it blocks this process, it basically means this process will now not get a chance to run till the pipe has more space that makes things more efficient .

So, in a way now sort and unique programs can now dance together . So, sort produces something, get suspended; unique gets a chance to run, uniques consumes; pipe gets empty, unique gets suspended sort runs again sort unique sort uniques . If the buffer file buffer was very small you will always have this kinds of a scheduling patterns sort unique sort, unique, sort, unique if the buffer was larger then it can absorb more things.

So, they can happen any arbitrary order; it can be sort for unique sort for unique unique sort whatever . So, this way of doing things is; firstly, it is more space efficient. Secondly, more time efficient; you do not have to write anything with this and it also

gives you an gives the operating system nice and insight into the scheduling the kind of scheduling that should happen for the system to execute at full throughput.

Student: Sir, does the operating system (Refer Time: 51:33).

So, it may or may not that is and that is just an optimization even if it does . So, the user can control the size of the pipe; there is a maximum limit that the size of the pipe can be; pipe on current operating systems are implemented in memory you never. So, on operating system does not typically use the disk to implement the pipe.

So, if its execute; if the pipe is being implemented inside memory, you are automatically bound by the size of the memory number one and actually much smaller than that because you want memory to be available further things too. So, pipes are typically much smaller than what you would have; what if you had implemented them in disks;

Student: (Refer Time: 52:19).

So what if the pipe is so small that even one write cannot complete properly; that is no problem. The semantics of write that till it writes entire data let us say this semantics (Refer Time: 52:36) write are till it tries to limit direct data it blocks . So, if you want to write 100 bytes and at 50 th byte it got blocked it just gets blocked and now the other guy the other process can should be somebody some reader should actually wake up if any and if it breaks up its going to consume.

And so the write has not yet returned and yet scheduling transfers are taking place . So, scheduling transfers are not necessarily demarcated by execution inside the application; scheduling transfers can happen in the middle of the execution of a system call.

Student: (Refer Time: 53:16).

Interesting. So, unique is giving its output to word count and the word count ; word count is supposed to compute the count of the entire file and if I just received half the word count; then what does it mean really a word count to ; word count cannot really work. So, the way word count program must have been written is that it is going to read the input, till it reaches end of input and then it is going to compute something on that input . So, till unique is producing something, word count is consuming something; word count will never produce anything till that has seen the end of input marker .

So, in this case unique; so sort is producing for unique and unique is producing for wc and these programs could be written either in either in online fashion which means I read few hundred bytes of your thousand bytes of a million bytes and then I can compute something on that or they can be written in a way that I want the entire input and only then I can compute something on that.

Student: (Refer Time: 54:32).

In their own applications space ; so they have allocated buffers in their own application space; they are going to put it in to their application space and now they are going to compute on it. Just like you write a normal program you; you read from a file you store it in an array and you compute something on the array; instead of reading from the file; in this case you reading from the file.

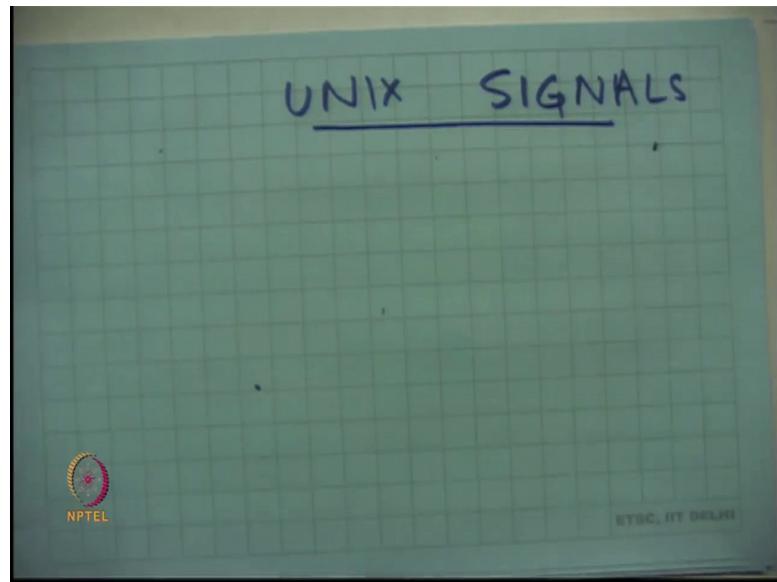
Student: (Refer Time: 54:57).

So, read will only return values that have ever been written to the pipe; it will never return any unnecessary garbage values and it will return the values in exactly the same order, in which they will return to the pipe. So, read will never return any wrong value really and if you say I want to read the next thousand bytes; till 1000 bytes are available let say the read blocks . Or it returns saying I have not; I have not been able to read I only been able to read 2 bytes and here the 2 bytes and now the it is; the to the programmer to say give me the next 998 byte .

How does the OS release the resources for the pipe? Interesting question; anybody? There is a system call that we discuss close . If you close both if you close the ends of the pipe if all the processes in the system, so a pipe is associated with the file descriptors of certain processes in the system . If you can figure out that all the ends of this pipe are closed so no process in the system actually points to this end of the pipe and the pipe is empty, then you might as well just completely release the resources.

Because if somebody calls read on that pipe you can just say that you can block it you can say nobody will write to it there is no writer to it. So, the pipe one end of the pipe is completely invisible to the system and so at that point you can release the resources for example, good the next. So, pipe is a very interesting Unix subtraction.

(Refer Slide Time: 57:07)



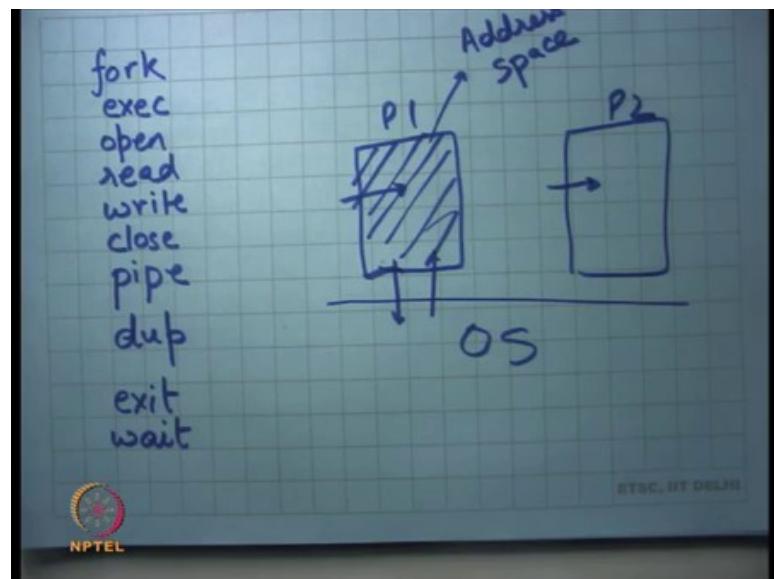
The next subtraction I am going to talk about are Unix signals and now we are going to do this in the next lecture.

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

Lecture - 03
Threads, Address Spaces, Filesystem, Devices

So, welcome to Operating Systems lecture 3.

(Refer Slide Time: 00:28)



In the previous lectures we have looked at Unix and its interfaces, in particular we looked at the system calls fork, exec. Fork was a system call which copies a process and makes a new process out of it, exec overwrites the current process with an executable and starts running.

Open, read, close, operate on files or other resources and use file descriptors as handles. Pipe is a way of creating an inter process communication channel between two processes, dup allows you to duplicate a file descriptor inside the file descriptor table. Exit is a way of telling the operating system that the process has finished and wait is a way of telling the operating system, that I want one of my child processes to finish. And, I want to get its return value or the exit code or the exiting process got it.

There is an operating system and various processes like P1 and P2 running on top of it. What is a process? Processes you can think of it as a running program, a program which

is an action which has state associated, running live state associated with it. The abstraction of a process says that the entire space, where it is running in main memory is entirely private to it.

So, if I allocate something inside the process it remains completely private to me, if I write something to in this space it remains completely private to me and nobody else can touch it. So, this space is also called an address space. So, this is an address space, it is called an address space because if I say I want to access address 1 million then the address 1 million here will point different places in p1 and p2.

So, the same address in different address spaces points to different locations and an address spaces is also an abstraction that is implemented by the operating systems. So, process also means that it will have its own address space. The nice thing about this kind of isolation between processes is that different processes can run completely independently and they do not need to be worried about each other and, worried about what they are going to access and whether I should protect myself with another process or not. So, as an example your web browser does not need to worry about a shell or your compiler other thing. They can all sort of execute independently because they are executing in completely independent addresses spaces.

Whenever they need something which requires access to shared resources and the shared resources could be devices, they could be files etc; they use things like system calls or even memory. So, if they want more memory anything; so, clearly you have one physical system that is being shared by all these processes.

If I want to get access to these processes these resources the process needs to ask the OS for it and the OS is going to decide whether this request is legal or not and appropriately arbitrate these resources among these processes. And, also ensure that concurrent accesses to these resources from these processes are protected.

And so, fork exec open are is one way of doing these things, fork is a way of creating a new process. So, process the list of processes is also in some sense a resource and if you want to create a new process you ask the OS can I want to create a new process. The OS checks if it has enough resources to actually support a new process for example, does it have enough memory to support a new process.

And, if it does it will create a new process and it will return the ID of the new process, if not then it will return an error code which will be a negative number. And, similarly if you open a resource like a file whether you have permissions to open that file or not. So, in some sense the OS is acting as an arbitrator between the process which is untrusted and the resource.

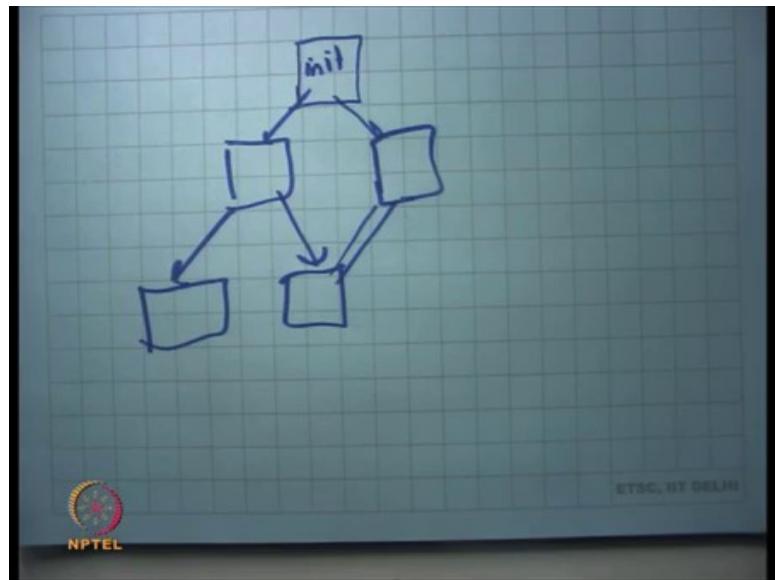
So, process can be thought of as an untrusted entity, it is a program that is running, you do not really trust it, in the sense that you do not trust it to the program could be greedy, it may try to take too many resources, it could be malicious, it may be trying to crash the system or it may be just buggy; in either case you do not want a buggy program to crash your entire system. So, it is a job of the operating system to ensure that all your resources remain safe even in presence of these untrusted processes. And so when you design these interfaces you also take one of the important things that you have to worry about is security. , you do not want untrusted processes to be able to take control of a system in unexpected ways.

And, another thing you have to worry about is performance and we saw some examples of a how different interfaces have different performance trade-offs for example, we looked at create process was this fork and exec and other things. As we go along the course it will become clear why this interface is reasonably performance and that is why it's popular. So, both performance and security are quite important when you are designing such an interface.

Now, the process P1 can communicate to the OS, but often a process needs to communicate another process. And, we saw some examples we saw one example last time we have one process wants to wants to compute something on its input. And, then pipe its output to another process which wants to compute something on its input and then and so on.

So, a chain of processes with pipes connected in the middle and this is a typical example of a situation which requires inter process communication and, pipe is a system call to be able to do this inter process communication efficiently al and then we have also looked at exit and wait.

(Refer Slide Time: 06:45)



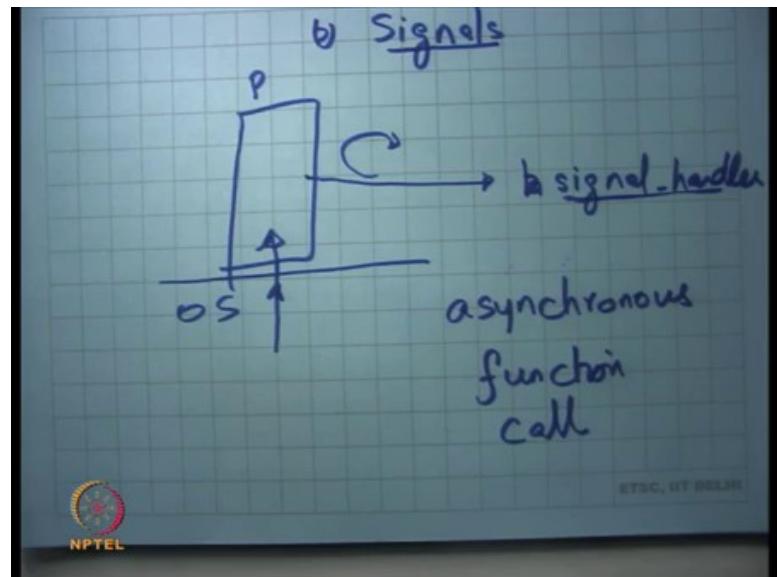
so, the processes in a system are often represented by a hierarchy which is the hierarchy in which they were created. So, for example, when you boot your system the operating system will typically create one process which is often called the init process. And this init process is going to fork more processes and more processes and these will spawn more processes in turn. And, also if needed the init process will create pipes between these processes and so, the system looks like a clear processes with pipes connected between them or inter process communication channels between them in general.

So, for example, when I when the system boots up it will spawn let us say a shell, it will spawn the x server, it will spawn let us say some default applications. It will tell the default applications that your default terminal is this shell or it will tell the default applications that your default terminal is on the x server and it will connect them appropriately. So, these things can run simultaneously and relatively independently.

So, you have many programs that are running independently and yet they can communicate with each other to do the things that you want them to do. I will point out here that pipes are one way of doing inter process communication and there are other ways of doing inter process communication which are pretty much similar in their nature where you make system calls to be able to write things to that you create a channel.

And, then you make system calls to be able to write to that channel and then you make system calls to be able to read from the channel.

(Refer Slide Time: 08:27)



So, that is a review of what we have done so far and today I am going to talk about another abstraction which are called signals. Often when a process is running you want to interrupt it and you want to say I want to do something. So, process is conceived as a sequential flow of instructions. So, one instruction after another is running that is the normal flow of execution of a process. But, let us say I want to bring to the attention of a process a certain event for example, a key was pressed. So, that is done using what is called signals.

So, a process is called a signal with a certain signal and the semantics of a signal is that it amounts to an asynchronous function call. So, let us say this is a process which is running, this thing means that its running and now the OS wants to raises the sign. What this means is, the semantics of a signal is that it will interrupt the current execution of the process and make a function call to the signal handler. So, whatever it was doing it suspends that in some sense and makes a function call to this special function which has been registered previously and makes a call to that function. And that special function is called a signal handler, that signal handler also executes in the address space of that process.

So, the signal handler executes in the address space of the process and can touch all the memory that the process can touch or see read and write all the memory that the process can read. When the signal handler returns the execution continues from where it was interrupted. So, that is the semantics of a signal, in some sense it is like an asynchronous function call.

Student: (Refer Time: 10:31).

Yes.

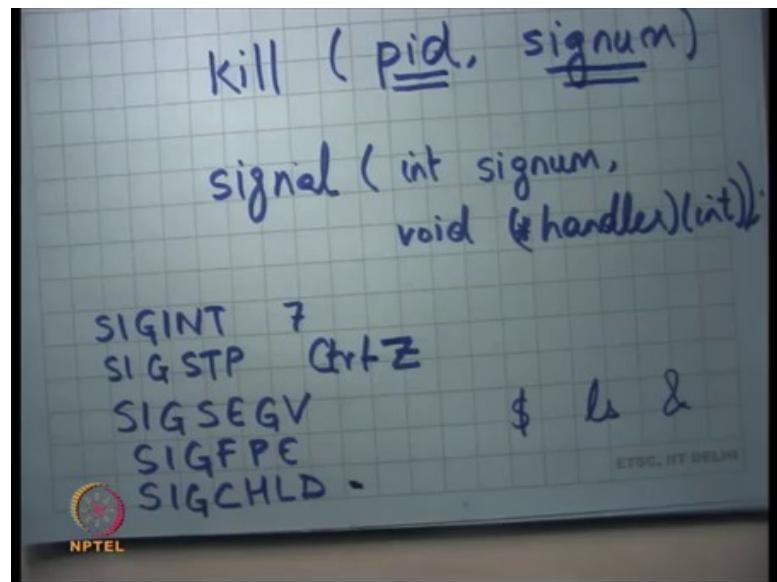
Student: Sir, you said that signal handler executes in the address space of the process. So, can it happen that the signal handler and the process that is currently being executed are totally different, they are not related. So, can that happen?

So, by not related you mean?

Student: We registered signal for process 1 and by scheduling policy suppose OS is running process 2 and then this signal came up.

So, signal handlers are private to a process. So, if process 1 registers a signal handler then while process 2 is running and a signal occurs, process 1 signal handler will not get executed; process 2 will have its own signal handler. So, every process will have its own signal handler and those signal handlers will execute in the context of its own process. So, this can be thought of as an asynchronous function call, it's basically like saying that whatever you were doing just insert a function call after that. So, you call that function that function when it returns, it just returns exactly where you call it and you can start executing the next instruction. So, the reason this kind of an abstraction is needed is to be able to handle rare events, events which occur which are sort of events to allow a process to be able to we can expose an event driven interface, like press of a key , arrival of a network packet or something like that. So, you want some kind of an event driven abstractions for a process to be able to do more than just sequential execution.

(Refer Slide Time: 12:21)



So, there are two system calls on Linux: one is called kill and another is called signal which support this Unix signal abstraction. Both of these are system calls and signal takes two arguments; the first is a signal number and the second is a function pointer to its handler. So, process can actually register handlers for signals. There are a set of standard signals that are pre-documented which are part of the OS semantics.

For example, there is a sig interrupt SIGINT, the interrupt signal which is triggered when you press a control C while the process is executing. So, if you press a control C what happens is the operating system figures out that control C has been pressed on the standard output of this process and so, it generates a SIGINT for that process.

What that causes is that, it causes an asynchronous function call of the SIGINT handler, the default SIGINT handler will simply exit, will simply called the exit system call and the process stops . So, that is why that is why you see that most of the times when you press control C the process just exits.

And, operating a process may choose to overwrite the default signal handler, the default SIGINT handler with its own handler for example, I just want to count the number of times control C was pressed on this while this process was executing. So, it can do that for example, or it may say whenever user press control C then free this data structure or start a fresh, it all these are possibilities.

Student: Sir, what does a process acquire default SIGINT handler?

That is just a part of the semantics. So, the Unix the operating system will say this will be our default signal handler and it will map it in the space address space of that process .

Student: Signals response is used to register the signal handler.

Yes, signal the signal system call is used to register a signal handler with a particular signal number. And, SIGINT has a particular number let us say; I do not know what it is, but let us say its 7 hypothetically. So, SIGINT stands for 7 and so with the signal interrupt number 7 the handler is. Similarly, there is sig stop which basically suspends the process and control Z, delivers the sig stop signal to the process.

So, I said the signal system call allows you to overwrite the signal handler, the default signal handler or whatever has been previously registered with a new handler. Now, the question is can I overwrite all signals? For example, there is a signal called sigstp which is which you can invoke by typing control C on control Z on the standard output. And, you and if the process is actually allowed to overwrite this signal then there is no a user will completely lose control of what the process can do.

So, there are certain signals that the operating system will allow you to overwrite and there is certain signals that you it will not allow you to overwrite. And so, six stop happens to be one of those signals that cannot be overwritten. So, clearly the signal system called as a process called the signal system call, the OS can decide whether it wants to allow this overwriting or it does not want to allow this overwriting. And, there are rules when it will allow and it will not allow.

Student: Sir, the integer argument for the handler function sir, that is the signal number itself.

Yes.

Student: Sir so, if you are passing a given signal number and we are passing a function for all possible signals when we call the signal (Refer Time: 16:24).

No. So, that handler is associated with that particular signal number.

Student: Sir, but it takes an integer arguments so.

.

Student: This is only for a just a single signal, then it need not take an argument.

. So, this is a simplification often what people do is use the same function to handle multiple signals. And so, this argument of this function allows you to disambiguate what was the signals and you may want to take different actions depending on what you do; so, it just a simplification .

So, the signal handler if you notice takes an integer argument which basically says what was the signal number that was generated and so, that allows you to have the same function pointer for multiple signals. So, the same handler for multiple signals and .

Then there is another signal which I am sure a lot of you have seen it is SIG SEGV, this is segmentation fault. So, even the segmentation fault is treated as a signal. So, what happens is if the process does something illegal which means that touches the memory location that it's supposed to not touch, the operating system will figure that out in an efficient way and we are going to see how it figures that out in an efficient way.

But let us say it is able to figure this out that the process is trying to access and address which it does its not supposed to access, then what it will do is it will generate a segmentation SIGSEGV for the process . And, now the SIGSEGV handler is going to get called and so, the default SIGSEGV handlers is going to abort the process .

And, then similarly there is a sig floating point exception for example, if you do a divide by 0 you will divide by 0 you are going to get a SIGFPE and similar things as SIGSEG V are going to happen. There is another signal called SIGCHILD, this is a signal which is generated if one of the child processes of the process of the current process exits.

So, let us say I started a child process and I wanted it to run concurrently; I did not want to wait on it, I did not want to block. So, I wanted for example, I start a browser window and then I start an; I start a new window from inside the browser window and I want both windows to exist simultaneously.

And, now one of the windows is closed which means that calls exit and so, you may want that the parent process be notified that the child has exited for it to maintain certain

data structures about what are all my outstanding children for example, or anything else I did not want to do.

So, SIGCHILD signal is a way of the process or the parent process to know that its one of its child children has exited and now it can check what has happened really. So, for example, in the shell example when we type what happened was ls was spawned as a new process. And, the shell implementation we saw so, far used to wait on that process to exit. If instead I did ls ampersand, I do not know how many of you have used this syntax of shell. It basically means that you spawn a new process, but you do not wait for it.

So, you also let the shell continue execution and let ls continue as a separate process, ls may not be such a natural example, but think of it as a browser for example . So, browser can continue on the shell and you want to type more commands on a shell for example. But, what you also want to do is that when the child exits you want to know what it returns status was.

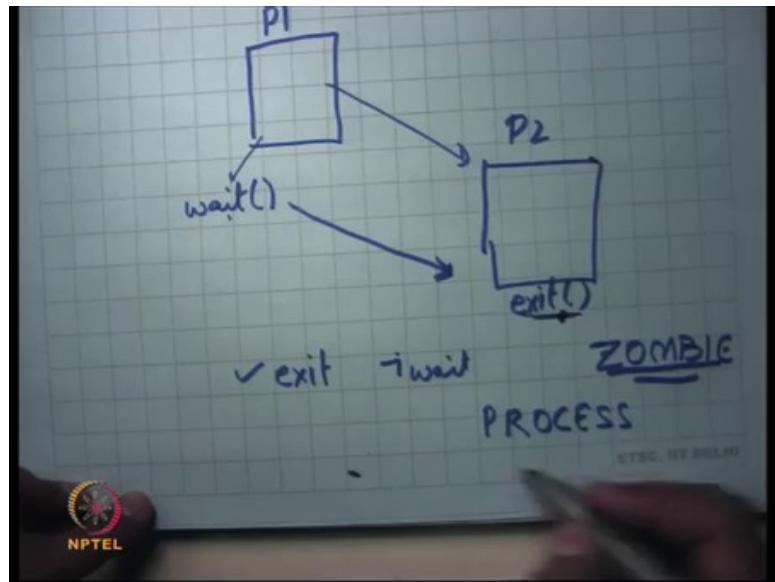
So, that is how SIGCHILD can help and finally, this kill system call semantics of the kill system call our that you can actually ask the OS to send a signal to another process. So, you can say that send the signal signum to this process pid .

So, one process can actually send a signal to another process by simply saying that I want to send a signal to this pid, this signum. Of course, there are rules I cannot send a signal to a process running by another run by another user, I can only send certain types of signals and so on . So, with those rules the skill system call allows you to do these things.

So, I do not know how many of you have used this program called kill which its the program goes rogue you just say kill dash 9 the program, that program is just calling this kill system call to and with the pid of that program to basically.

And, the signal that sending it is from SIGINT or sig kill or something sig kill and 9 stands for sig kill. So, that is an introduction to Unix signals, it allows you to handle interrupt based or signal based computation which is relatively rarer and which requires event driven handling to be doing efficiently .

(Refer Slide Time: 21:57)



Another interesting thing about processes is that let us say I am a process P 1 and it called one other process P2 and the process P2 finally, calls exit . And, while the process P2 has called exit the question is should the operating system delete all state which corresponds to P2 or should it hold on to the state for some more time? And, the reason I ask this question is because P 1 may actually call wait sometime later and wait is supposed to return the exit code of the process P2.

So, if exit of the child was called much before the parent called wait then its the responsibility of the operating system to preserve the exit code of the child. So, that when the parent calls wait it can return the value. So, in some sense a process is not cleaned up just after exit, a process is truly cleaned up after its parent calls wait on it.

Till the parent does not call wait on its child process, there is some state of that process that remains in the operating system. And, now a process which has called exit, but its parent has not called wait it is called a zombie. So, not wait, but exit it is called a zombie process.

Now, one easy way to remember this is it is like saying that somebody has died, but his “aatma”(english meaning is soul) is remaining because nobody is called wait on the [FL] or [FL] has not been done, it like we talk about it. So, that is a so, its zombie process the return code of the zombie process is actually just lingering around in the orient system and this kind of bug is actually very common.

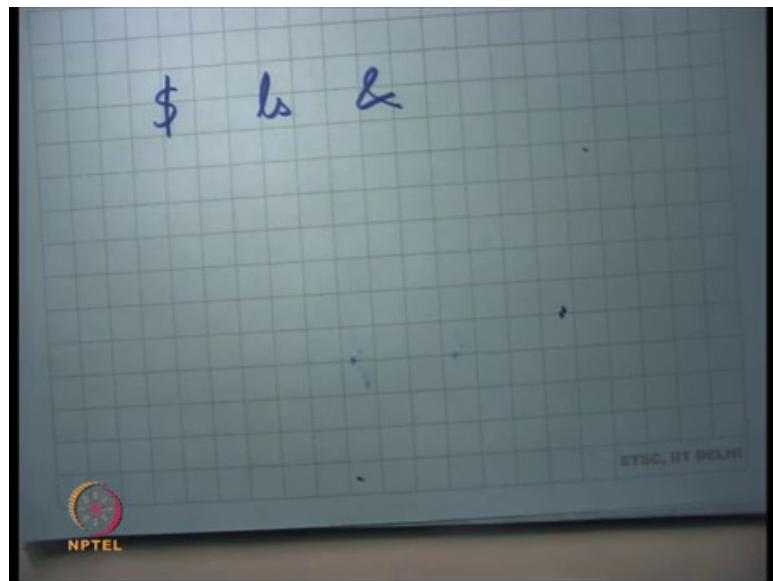
Because, what happens is that the parent the programmer forgets to call wait on its child and these kind of this is called a process leak or so, it is called a leak basically. It is a leak because you created some process some number of processes and you forgot to call wait on it. And so, these processes keep occupying some space in the operating systems structures which store its exit code and they will remain forever.

And, if it is a long running system then you can imagine that over months you are eventually going to run out of memory space, storing these things has a question.

Student: Sir, in the excessive bo it says that you wait call returns the pid of the exit child which was died.

So, x v 6 and Unix are slightly different number 1, the syntax of wait may be different, but the return code is definitely returned in the wait system call . So, I believe there is a pointer that the wait system call takes which basically gets stuffed with the return code. So, there are two ways; so, there is a return value of the wait system call and then there is a pointer that it takes where the operating system is supposed to stuff the return value. So, in any case the return value needs to be preserved till the wait system call is call.

(Refer Slide Time: 25:48)



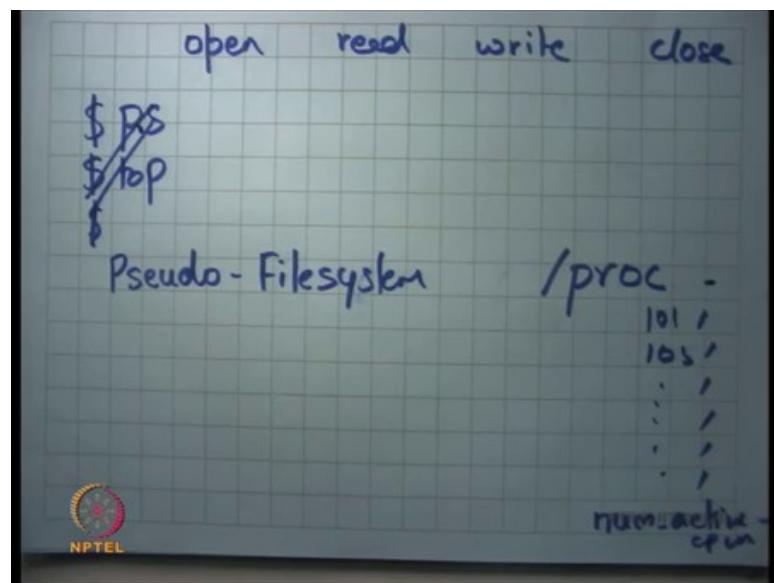
So, for example, on the shell if I say ls and if I implemented my shell by saying that I am not going to call wait on the child process, I just spawn the new process and I forget

about it and then I take the next command. What is going to happen is that will finish it will call exit, but it will remain in the system for ever because it will remain a zombie.

What is needed in this case is that the shell should, one way to do it is the shell code keep checking periodically whether all the children that I have spawned so far has anybody of them exited. Question is how often should I check? Should I check every 1 second? Should I check every 1 minute, 1 hour, 1 day? No, that is completely configurable, the other way to do it is to use signals . So, signals allow you to sort of more elegant way of doing things.

When you say that if a child exits, a SIG CHILD gets generated in the shell, the SIG CHILD handler checks all my checks which of my children have exited. And, if somebody has exited it calls wait on it, collects the exit status, does whatever it wants to do with exit status; perhaps it just wants to ignore the exit status whatever carries on.

(Refer Slide Time: 27:11)



Next, I want to talk about open, read, write, close and the power of these system calls. So, so far we have looked at open read write close in the context of a files and in the context of a devices and things like that. But, actually modern operating system use this system calls to do many more things.

For example, if I wanted to find out what is the number of processes in the system or if I wanted to find out which process has how much memory allocated and in which address

spaces . So, you can imagine that in one application typically may be interested in huge amount of information from the operating system.

For example, if you have ever used this command called PS it basically lists all the processes and what are their process IDs, what are the dependencies with other processes and how much memory they are consuming. So, there are other programs like top which tell how much CPU they have used, how much memory they have used etc.

So, a process may be interested in lot of information from the operating system, also a process may want to configure the operating system at run time in several ways. For example, then operating system can provide the functionality that you can on the fly change the number of running CPUs in the system. For example: you want to do some kind of energy saving computation and you want to say I only want the number the processor only I have 16 processors in this machine.

But, I only want the processes that I am using to be on everything, else can be off and you want an application to be able to control that let us say. So, an application to be able to say oh just switch of 10 process processors and I just want the remaining ones do work. And so, the you can imagine there is a plethora of different functionalities that an operating system is burdened with to provide to the user.

And, another system called we have seen so far really solve any of these. We have been talking at a very macro levels and all these different things are much more micro level. And, the question is how many system calls should have an operating system provide to be able to get this kind of functionality?

Seems very daunting. The way this is done in on Linux for example, is using a pseudo file system called /proc. So, what this means is there is a the /proc lives in your regular file system prefixed with the slash character, but it is not a real file system, it is a pseudo file system. /proc will have subdirectories which will be all the process IDs for example, /proc /101 103 and so on. So, these are all the process IDs and these are all sub directories. And, what an operating, what an application needs to do to be able to for example, if I were to implement PS, all I need to do is open /proc and read its contents. So, the open and read system calls can be overloaded to be able to do any information gathering from the OS.

So, these all the information that the OS wants to wishes to expose to the to its applications can be made part of this pseudo file system. And, an application just needs to use the regular open read write closed system calls to be able to read those system call, read that information.

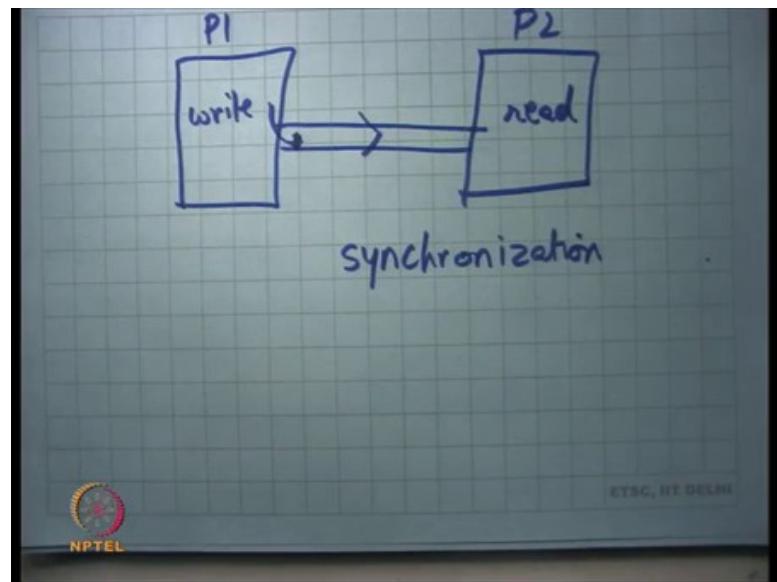
Of course the operating system can still interpose at an open call to see whether an application is indeed authorized to read this information. For example, I should for example, some of the simple rules are I should only be able to read information about my own processes and not allowed any other users processes and so on.

So, once again those kind of access controls can be made can be enforced at the open system call for example, or the read system call for that matter. Similarly, if I wanted to change the number of active CPUs that is also possible the let us say the slash proc has this file called num CPUs num active CPUs.

And, when a process says write to num active CPUs internally the operating system is going to trigger the procedure to actually switch off the other 16 minus 2 CPUs for example. So, you have overloaded the file system abstractions to do other things which are which involve getting and setting values in the OS itself. So, as an example I mean just to put things in perspective an operating system a full fledged operating system like Linux would have roughly 300 system calls al, that is still a lot.

But, it is still not in thousands or millions. So, far we have looked at processes how processes have private address spaces, how processes have abstractions to access resources that are provided by the operating system. These resources could be hardware resources, files or its own data structures for that matter. And, we have also looked at how different processes do inter process communication using pipes.

(Refer Slide Time: 33:30)



So, let us take an example of two processes which are trying to do inter process communication P1 and P2 and let us say there is a pipe. So, let us say this is the producer and this is the consumer. So, he is going to call on this pipe and he is going to call read on this pipe.

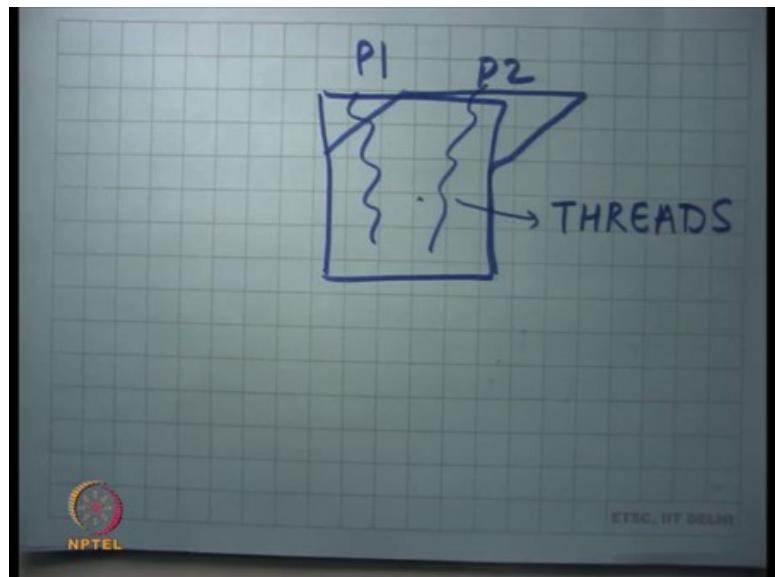
This pipe also acts as a synchronization mechanism because, if P2 calls read and the pipe is empty it basically means at P1 needs to run some more and call some more writes before P2 can execute. So, in some sense they both synchronize with each other.

Synchronization means that they sort of communicate with each other and restrict their behaviour based on the behaviour of the other, that is one we are doing synchronization. This way of inter process communication is slightly expensive in the sense that to be able to write you need to make a system call and we are going to see later what is the cost of a system call.

So, a system call means you have to tell you have to make a function call inside the OS which means the OS has to do a few things, get your arguments, process them. Then the other the receiver has to call and make another system call read and those arguments need to be given to the into the address space of the other process. So, there is some amount of overhead, there is a copy that is involved from the address space of the first process to the operating system kernel.

And, the and then a copy from the operating system kernel to the address space of the second process and these are relatively expensive operations.

(Refer Slide Time: 35:11)



On the other hand if it were possible for two different processes to share the address space, you could have done this communication much cheaper. For example, if I want to send a data or message from P 1 to P 2 and we are sharing memory; all I need to do is set a byte in my shared memory and the other one is going to get that byte .

So, read and write two shared memory, the kernel is completely out of the picture in this case. And, you can imagine write in this case is just a memory write and read in this case is just a memory read.

And, both are significantly cheaper or more efficient than doing write and read system calls in a pipe. So, this now this scenario where two different processes shared the address space, these entities which these are the sort of different entities that are existing simultaneously, but they are sharing the address space are also called threads .

So, a thread is an execution flow; so, a thread governs an execution flow and two separate threads can have different execution flows. A process is a thread plus an address space. So, a process basically means it's an execution flow with its own private address space. As opposed to a as opposed to two threads within the same process sharing the same address space.

What are some advantages of using processes and threads? Well we have already seen the advantages of processes, you could not have different processes running different programs which are completely independent of each other. One process could be waiting on could be running on the CPU another process could be waiting for the disk to get some data yet another process could be waiting on the network. And so, that way you have full utilization of your hardware resources.

So, for example, I have three processes, a browser, a compiler and a shell, the shell maybe just waiting for the keyboard input which means it is just blocked, its not it does not need to run on the CPU until user presses a key; the compiler may be actually running on the CPU and so, it is using the CPU, its keeping the CPU busy. So, while the first process is also running it's actually the OS has been able to multiplex these two processes in a way such that CPU is use in the most efficient way.

And, yet let us say the browser is waiting on the network card to receive the next packet and so on. If I did not have these this process abstraction being able to implement this, this multiplexing would have been much harder . Because, I would have had to have one program that understands that this part of the program is now waiting on the network and this part of the program is now execute on the CPU.

When you split into processes the and expose system calls to the processes the operating system has full visibility that this process is actually looking for the network. And, this process is actually executable CPU and that is where you have more utilization.

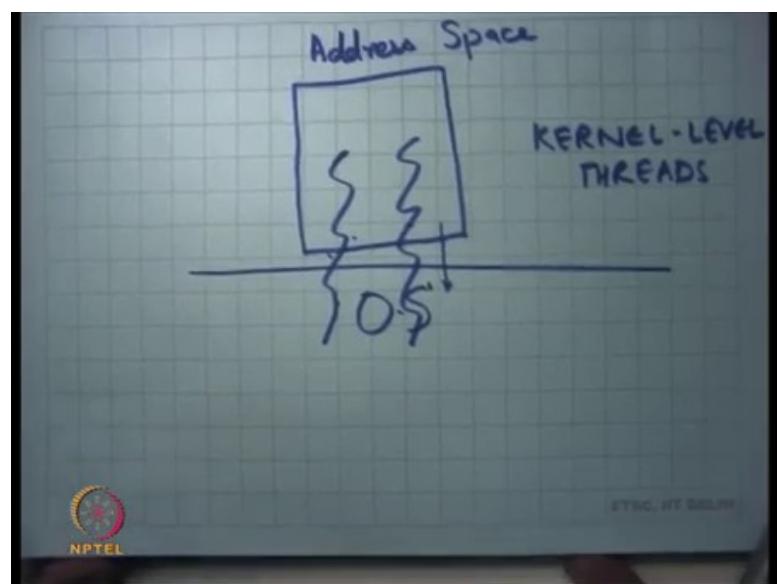
Further, when you talk about multi-processor systems the only way you can use other processors if you have multiple threads of control. If you just have one thread of control that is running then the other CPU will never be able to run. So, you need multiple threads of control to be able to actually utilize different processes, processors in your system. So, you need multiple processor processes to be able to actually keep multiple processors busy.

Same thing with threads; so, threads give you the same kind of advantages. You get more concurrency, you get more utilization out of your system. One thread could be waiting on the network, yet another thread could be executing on CPU 1 and yet another thread could be executed on CPU 2, all simultaneously to give you the maximum system

throughput. The advantage of threads over processes is that because they share address space they can have very fast inter process communication.

The disadvantage of threads over processes is that because they share address space there is no isolation. So, whatever one thread does the other thread is not protected from it. So, in other words two threads need to trust each other; what is more they need to be designed each such that they should be know they should know the existence of the other.

(Refer Slide Time: 40:30)



For example: the fire for the browser and the compiler do not need to know about each other because they live in separate address spaces. But, if you made them threads then they will have to know about each other and do appropriate safeguards for doing proper protection.

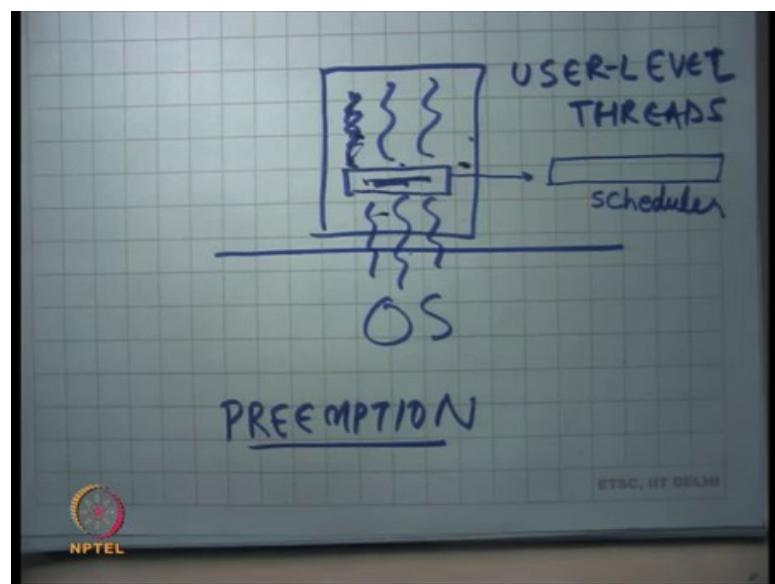
So, how does one create threads? Let us say here is a process and the instead of calling it a process I am going to call it an address space. So, here is an address space, we saw that this fork system call was actually spawning a new address space and also creating a new thread of control.

What we now want from the OS is to be able to create a new thread of control without spawning a new address space. And, the only way you can one way you can do it is to have another system call which creates a new thread. So, you could have multiple

threads within the same process, within the same address space and you would need to have some system calls on your OS to be able to do that .

If you do something like this then the operating system is aware that these are independent, independent threads of control. And, one of them can execute on one CPU and another one can execute on this another CPU provided the same address space gets mapped on both CPUs, the same memory gets mapped on both CPUs.

(Refer Slide Time: 42:00)



Another way of doing threads or of implementing threads could be that the OS has no idea about multiple threads. So, what you do is you tell the OS that I am only one process, but internally you have some kind of a system which allows you to emulate multiple threads of control al; there is a question.

Student: Sir, in this if I want to exit one of the processes and I cannot actually because, OS is that there are actually two processes running in the same addresses (Refer Time: 42:32).

So, in this scenario you are saying if I if one thread wants to exit then it will not be able it will cause everybody else to exit also, that is what you are saying yes true. So, so just one so, what I am trying to show is that the two ways to implement threads: one is to tell the OS to create multiple threads of control within the same address space in which case

threads are a first class entity. And, the OS knows about them and as OS is scheduling them across the multiple resources that you have in the hardware.

Another way of implementing threads is to not tell the OS about it, but to be able to do it internally. And, this is interesting because I am going to tell you next about how this can be done using the abstractions that you have studied so far. As you can imagine the OS has no idea that there are actually multiple threads being emulated within the process. And so, if one thread calls exit it basically means the entire process gets wiped out.

Further, if one of these threads called the system called read and let us say the read was on its doing a read to the disk and so, the all the threads will now get blocked because one thread calls read, its as though the entire process called read . And so, there is no real concurrency and so, the hardware resources are not being used to its fullest. Similarly, if one thread calls and read on the network then the entire process gets blocked and so on.

Student: If the abstraction is kind of one way that abstraction could have been used for some system false also, that read or exit could have been blocked and modified.

. So now, the question so one way to do this is you wrap the exit system call into your own system call which is called thread exit. And so, the thread exit will basically just be stopped within the thread and that is going to just free up data, all data structures related to this thread and the other two threads can still remain .

So, that handles things like exit, but what if I wanted to do a read from the network that still needs to go through OS because, no abstraction provides you a way of doing read directly so far. I mean the abstraction we have discussed so far do not allow a process to do a read directly from within the process. You have to make a system call to the OS and the OS is going to say this is the whole process that has made a system call and so, block the process till there is a packet from the network.

Student: if we do not need concurrency in this case then we can divide times of the running.

. So, clearly firstly there is really no concurrency in the system. Where is the concurrency? You can only run on one CPU, you block on a resource everybody gets logged on the resource. So, there two threads can physically never run concurrently, but

logically they can be made to look like they run concurrently and that can be done by just for example maintaining a queue in some sense maintaining some kind of a scheduler inside the process.

And, the scheduler is going to say it is going to create multiple threads and it is going to say this thread gets to run, that thread runs. And, then when it yields then you bring it back and we say now this thread you get to run and then when it comes when it says I want to yield, it yields and then you say another thread gets to run and so on. So, this is a way that a process inside itself is implementing a scheduler in sort of emulating multiple threads.

Student: Sir, what is the benefit of running (Refer Time: 46:24)?

Let us come to the benefit a little later, let us just understand how this is how can this be done using the abstractions; I want to talk about the benefits so. The other thing that an OS does is a process should not be able to run away with the CPU. So, once a process has been scheduled on the CPU and OS should have a way of taking it back from the CPU. And, the way this is done is through what is called an interrupt.

So, if a process is running and an interrupt occurs the interrupt handler is the OS interrupt handler. And, the OS interrupt handler is going to say this process has been running for too long, let us just suspend it and get let some other process run on it. A similar method can be used at the user level using signals.

So, a user process can register a signal and it can say that the signal; so, there is another system call called alarm which allows process to tell the OS that a signal sig timer let us say should be generated every 100 milliseconds. So, every 100 milliseconds the sig timer hits a signal is going to get generated, the sig timer handler is going to get called inside the process.

The sig timer handler is maintaining this data structure of all the active threads in the system. It takes the running thread, suspends it, puts it in the ready queue, takes another thread, sets it running and returns. So, that way you also implement what is called preemption.

So, you can preempt a thread while its running and you can emulate a scheduler inside the thread. So, in some sense if you look at the abstractions we have learned so far the signals abstraction of Unix is paralleling the interrupt abstraction that you see in hardware.

So, a hardware interrupt like a keyboard, key press or timer interrupt coming in or a network packet coming in these are all like activities that occur asynchronously and relatively rarely. And, you want to be able to and an OS is supposed to handle them and the way to handle them is what is called interrupts. So, an interrupt comes and the OS handles interrupt handler gets called, similarly inside a process a signal comes and the signal handler gets called.

Now, the question is what is the advantage of doing something like this? Well, clearly there is no concurrency achieved by having multiple user level threads inside. The advantage of kernel level threads, this is actually giving you physical concurrency.

The advantage of user level threads is that context switching is much cheaper for user level threads. If I want to context switch from one thread to another, let us say one thread says I want to give away the CPU, I want to yield. All it needs all that has happened is it will make a function call internal to the process and the process is going to take that, put it somewhere and set another one running.

So, there is no kernel crossing, there is no system call that is required which as you are going to see is relatively more expensive. So, there is you can do more things at the user level and you can do them faster. Further you can map multiple user level threads to multiple kernel level threads. So, you can the scheduler could be made intelligent enough such that you basically create multiple kernel level threads at the bottom. And, let us say you have two kernel level threads and you have five user level threads and its like saying that I have two processors and I have five processes and then I can map them anyway I like. And, I can have a completely custom scheduling policy depending on my application.

So, a kernel scheduling policy it has to be very general because it is going to run a variety of applications. A user level scheduler can have a very very custom scheduling policy which suits that vertical application. And, more importantly often software is written in a certain style and you want to be able to run it in different environments.

If your environment happens to be such that the kernel does not support, kernel level threads that is that used to be relatively common, still quite common. Then one way to be able to run the same program on in this environment is to be able to abstract it as a user level thread. The same code can run on this library where you are actually fooling the program to say believe that it has threads whereas, its actually running on a single thread, single kernel thread.

Student: And, during emulation also maybe emulating a different hardware also.

Sure. So, anything which involves running code that was written on a for assuming certain thing, but running in a different environment you can use this kind of a wrapper layer to be able to do these things .

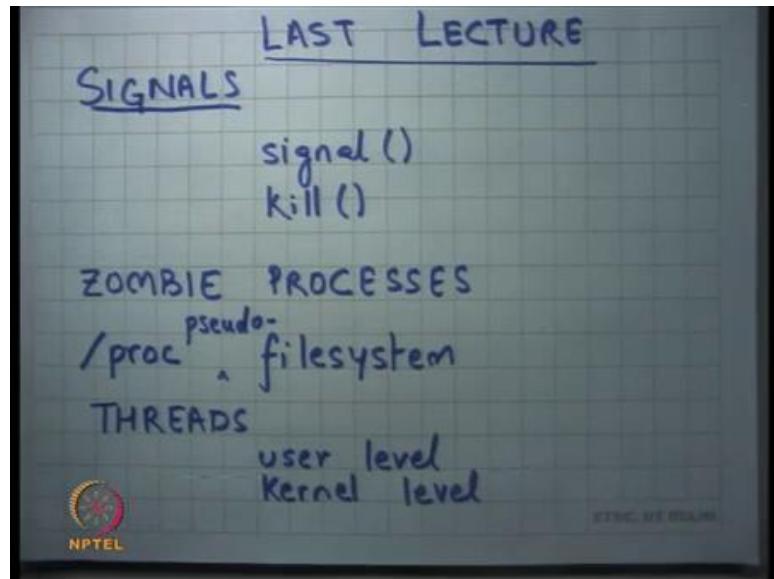
So, we will stop here and continue next time.

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

Lecture – 04
PC Architecture

We welcome to Operating Systems lecture 4.

(Refer Slide Time: 00:33)



In the last lecture we looked at Unix signals. Signals is another way of doing inter process communication or communication from the OS to the process as we saw. And it is useful if the communication indicates a rare event, a relatively rare event for which the process may not be ready. And so, it is delivered to the process in a very asynchronous fashion.

You can think of it like there is a process that is running, and another process wants to bring something to another process. So, process p1 wants to send some message to process p2 and process p2 may not be expecting that message or process p2 just does not think that it is worthwhile to keep trying to poll for that message every few you know at pure time intervals.

So, one way to design your processes would be that you basically let install a signal handler in the receiving process. And, in the sending process you send something; let us

say you send something to the pipe and then you send a signal into the receiving process, the signal handler of the receiving process is going to read from the pipe. And so, in that sense you have an event driven message transfer between the sender and the receiver. And the same concept can apply from for communication between the OS to the process and various signals we saw last time; like interrupt segmentation fault etcetera are ways of doing it.

So, it is another way of inter-process communication which is useful in settings where the communication is relatively rare and it needs to be handled in an event driven fashion, ok. So, we saw two system call; signal which install the signal handler and kill which allows you to send a signal to another process. So in fact, when you for example, type control C on the shell what happens is, that the shell passes the control C command and sends a makes a kill system call to send SIGNIT to the process that is currently running in the foreground of that shell, alright.

We also looked at you know we also talked about this concept of zombie processes. So, we have seen that the two system calls as exit and there is wait; a process can call exit, but whatever is the exit code that is returned by the process should be communicated to the parent and the parent can read that through the wait system call, right. Now there is this small dilemma that if the process has exited and the parent has not called wait, what should the OS do, right.

The OS has no idea when the parent may call wait; the parent may call wait one second from now, the parent may call wait 10 years from now or the parent may actually never call wait, right. So, all these are possibilities and the OS should maintain its semantics and all these possibilities. So, what the OS needs to do is it needs to reserve, it needs to store this return value somewhere in its data structure, so that when the parent calls wait it can serve it to it, right. And so, this extra state of storing this information of the process of the exited process, a process that is no longer living is extra overhead on the OS.

And so, these kinds of processes which are not really living, but they are some state of that some of the state is still living in the OS are called zombie processes, right. And we also said that zombie processes are a very common programming bug, because programmers often forget to call you know wait in the parent, they just spawn a process something exits. And what happens is eventually although the state is very small,

eventually the state is going to keep leaking and so, some at some point in future your OS data structures may actually find very little space for themselves, alright.

Then you also talked about the slash proc pseudo file system, here is an example of a file system which is actually not a file system, it is not it does not have real files, it actually just shares name space with the file system; which means it has you know it has a name which nests inside the file system namespace. A process can call open read write close on this on these names, just like it can call open read write close on names of files. And the OS can use this file system to expose its own data structures to the process, right.

So, for example, the OS can expose the list of processes or you know that process the concerned resource consumption of the processes or even allow the application to actually set values in the kernel in the OS's data structure, right. I have been using the term kernel and operating system interchangeably, they are mean roughly the same thing in this discussion. It is the kernel is the kernel is the real operating system layer that is implementing all these system calls ok.

So, we looked at the slash proc file system and then we also looked at and then you also said that, you know we talked about processes which are relatively isolated. And, we understand the utility of having processes; because one program does one program writer does not need to be worried about another program writer and you also have trust boundaries, one program does not need to trust another program and so on.

But then you know inter process communication becomes relatively expensive, because it involves system calls and going through the kernel. Often you do not need this level of protection and so, what you could instead do was, instead of use processes use shared address space and allow multiple threads of control within the same address space. And so, these are called threads, right.

And so, threads can communicate between each other in a very fast way, because they share memory; one thread can write another thread can read and so, it absolutely involves, does not involve the kernel at, alright. Some examples of applications where you use threads; let us say you have a browser and you open multiple tabs inside the browser, you know all the code of the tabs, each of the code is actually doing separate things simultaneously.

But this code has sort of mutually trusted, so you do not really need to have separate silos for each tab, they can actually run in the same address space that way they consume less memory number 1. Number 2 if they need to communicate with each other for whatever reason let say statistics, then they can do it in a very fast way.

And, but what this imposes a burden on the programmer is that, the programmer should be aware that when one thread is running, another thread could become concurrently running on the same memory. And so, there should be safety in that sense and we are going to talk about what kind of safety and how it is ensured later and that is a very interesting topic in itself.

We also said that there are two types of threads as user level threads, where the operating system is completely blind to what is happening; it is actually just looks at a process. But, inside the process the user can actually orchestrate his logic in such a way that it appears that there are multiple threads and each thread can now appear to run concurrently, right.

There may not be physical concurrency, but there is logical concurrency when you are doing user level threads, alright. And we also said that actually the, you know the system calls that we have discussed so far are enough for the user to be able to implement this layer of abstracting one process into multiple threads, right. And this will be part of your homework problem next week. So, you will see how this is done for example, alright.

Then the other type of threads are kernel level threads, where you tell the OS that yes I need threads and you know the OS understands threads; basically gives you one address space and allows you multiple threads of control. And, that allows you full physical concurrency, if there are multiple CPUs and these threads can be scheduled simultaneously physically. And also, you know one thread could be waiting on the disk and another thread could be waiting on the network and yet another thread could be working on the CPU, alright. So, we saw all this, yes question.

Student: When you switch off the machine does the zombie processes get killed?

When you switch off the machine do the zombie processes get killed? Yes, right. So, the semantics of an operating system typically are that when you switch off the machine all memory state is wiped out, all process state is wiped out, it is only the discontents that

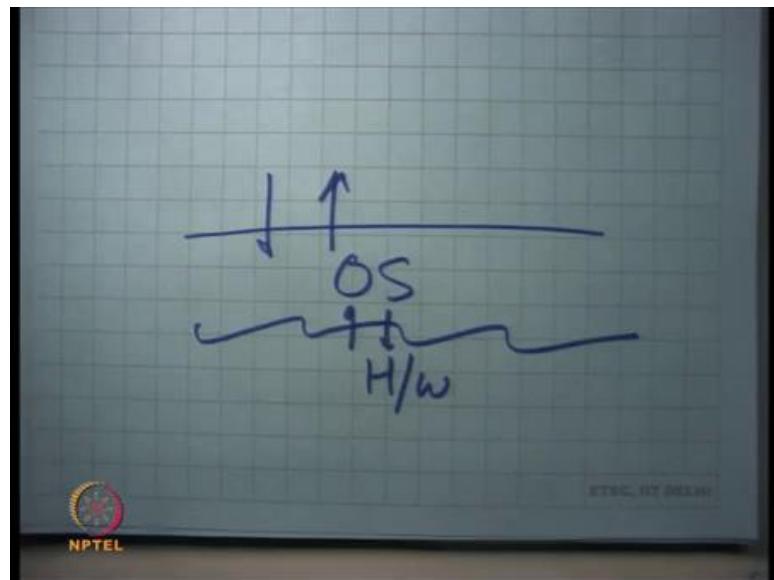
remain. And, we are going to look at you know even the discontents the that we guarantee on what remains and what does not remain and we going to discuss that when you are talking about file systems question.

Student: In the event when parent does not call wait for child and parent exits before child.

Interesting question; so, in the event that the parent does not call wait on child and the parent exits before the child has exited called exit, then what should happen; should the child remain as a zombie process or should it be freed completely, right. So, I mean, so you know an OS typically what it does is if you know, if a child loses its parent then it is an orphan process and that orphan process is actually attached to one of the default processes it is called the INIT process and so, you know if an orphan's process parent is the INIT process.

And so, the INIT process is you know, the code of the INIT process typically would just call keep calling wait over and over again to clean up all the zombies, right. So, that is one way of handing this, alright.

(Refer Slide Time: 09:09)

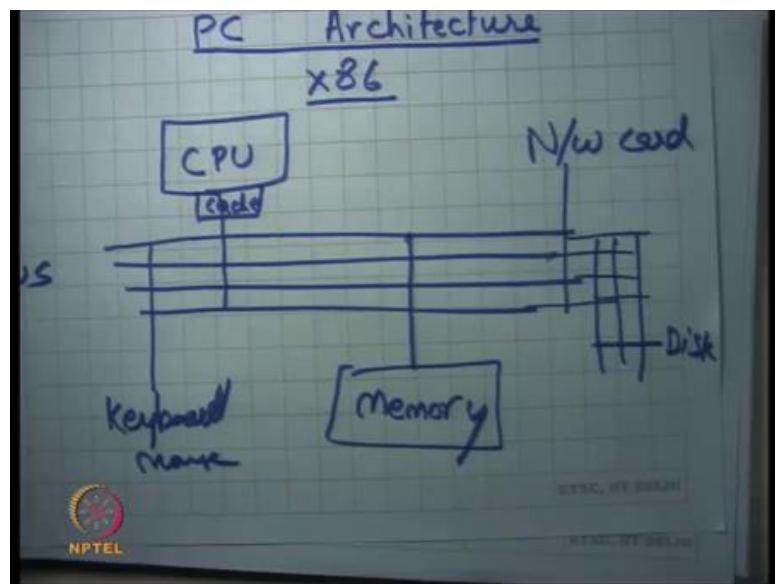


So far, we have looked at the OS and we have looked at what kind of abstractions does an OS provide? Now I am going to change gears and I am going to look at how does the

OS provide these abstractions and we are I mean it is going to be a relatively longer discussion.

But we are going to start with looking at what kind of interfaces does the hardware provide to the OS, right. So, let us look at the other side of how the OS works, you know how. So, let us start by looking bottom up and let us look at what the hardware provides, right.

(Refer Slide Time: 09:45)



And I am going to start, so we are going to have a discussion on the PC architecture or PC hardware. And in particular I am going to discuss the x86 architecture. Now there are many architectures, the many machine architectures that have evolved over the years and Intel's x86 is one of the very popular architectures. And, the reason I discuss it in this class is, because your programming assignments are based on this architecture and moreover most of our machines are actually that we use based on Linux or windows are usually based on this architecture. So, it is also a good practical experience of what the x86 architecture looks like.

As a word of warning the x86 architecture is very complex, much more complex than it needs to be, so we are going to slowly dig through that complexity. But I would like you to pay attention, because so that you understand this stuff and that will literally help you through your programming assignments for example, alright ok. So, let us first understand what a machine looks like, what a computer system looks like. So, let us say

here is the CPU, here is the memory, a CPU has a cache let us say, right and let us say this is the BUS, alright the system bus basically. And, then there are you know other buses attached to this BUS, which are let us say attached to devices which are like disk or you know network card and so on, right.

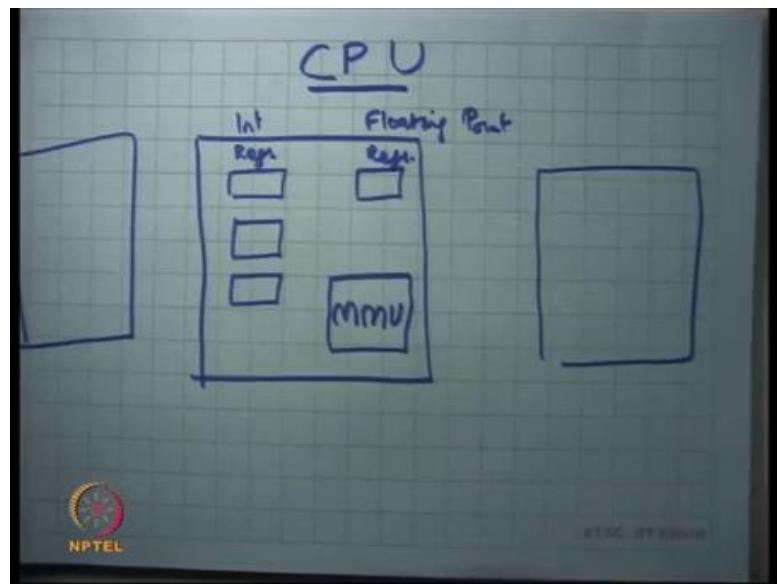
So, a CPU is basically a processor, it is like you know it is a silicon chip that has circuitry built into it. The nice thing about having a small silicon chip is that, you know operations within this chip are very fast, right. So, you can read and write within the chip in a very fast way, you can execute logic inside the chip in a very fast way, right. So, you have all these gates, all these logic gates inside the chip which are executing things that you want to execute. And, then you know there is the bus, this is this is a high-speed bus; you know that is a 10 to 100 gigabytes per second, it is connecting to memory.

And so, the chip can only have that much storage space, because you know that is the physical that is a limitation of the physical technology. And so, but if you allow slightly slower storage, which is you know typically the kind of technology used for physical memory, technologies let us say called DRAM; then you could actually, so you could have larger sizes of this memory. And these sizes could range anywhere from you know a few 100 of MB's to now even few 100 of GB's for example, right. So, memories are actually constantly increasing size in a huge way.

And so, the bandwidth to the memory is actually dictated by this bus. And so, the CPU can actually say that, can send a message to those memory on the bus saying give me the contents of the address x and the memory will reply on this bus saying here the contents of address x, alright. Or so, this is the read operation, write operation similar, the CPU you can say chain the contents of address x with the data D. And so, you know and then there are the memory has certain semantics that if you write to address x data D and you read it later you are going to get the same value that you wrote.

So, those are semantics of memory and they are implemented in a fast way using different technologies, one of the popular technologies doing data alright. And similarly, there is a device, so the CPU has ways of sending commands or device and the device has ways to send results back to the CPU, alright ok.

(Refer Slide Time: 13:51)



So, now, let us look what is inside the CPU. So, let us look at the CPU, alright. So, what does the CPU look like? The CPU has some integer registers, right. So, these are let us call them registers; what are registers? Register is just storage inside the chip, right; it is very fast storage, much faster than DRAM, ok. You can access it as at less than say nanosecond latencies, alright.

Then you have you know let us say these are integer registers and then you have floating point registers and you have you know a memory management unit, MMU. What is an MMU? An MMU is basically an instruct, so the CPU wants to access memory; the memory management unit sits between the instruction that wanted to access memory and the actual physical memory, and does some sort of bookkeeping or translation or access control checks, right.

So, you know an instruction wants to access certain name physical memory, the MMU is going to do some checking or some translation before it actually sends an address to the bus, right. So, some translation between what the instruction says I want to access and what the actual address goes on out to the bus for the memory to read, right that is MMU, alright.

And then you know modern CPUs actually have multiple of these on single chip, right. So, you know when you hear about multi core architectures or you do dual core processors etcetera, it is one chip which has multiple of these inside one. And, now you

also have some logic to basically be able to communicate between these processors. So, I am going to skip that for now and let us just talk about a single processor I make things simple, alright.

Of course, in this picture you know what else remains is basically you know, there is let us say keyboard and mouse etcetera. The way typically you would organize the system, is you look at the bandwidth requirements of that particular device. So, the bandwidth requirement memory is typically very high, because you may want to read you know megabytes of data advance and you want that to happen as fast as possible. Bandwidth requirements for keyboard are relatively very low, because you know how off, how frequently can I use a really press a key; it can at most be you know at millisecond you know, one key per millisecond at most.

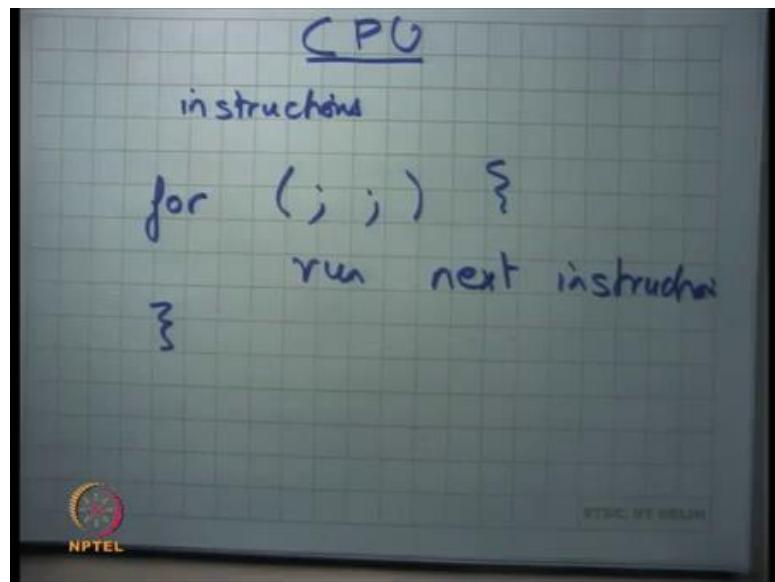
So, you do not need that much bandwidth to the keyboard. So, you would probably have a very thin bus to connect to the keyboard or mouse for that matter. For something like a network card, you know it really depends on what kind of network technology you are using; you know earlier network technology used to be let us say 100 megabytes, so the bus inside the CPU did not need to be that fat to connect to the network card.

Now, you have 10 gigabits or even 40 gigabits Ethernet, so you know the buses inside the CPU also need to be broadened appropriately. The disk, the disk typically you know is mechanical device you know when you actually see the disk moving, it actually moving a spindle; I am going to talk about how a disk works later.

And some mechanical devices do not have that much throughput anyways, so you know typically they you do not need that fatter bus to a disk let us say. So, I mean a hardware designer is going to do all this; engineering is going to say that ok, this is how I am going to maximize my throughput for the devices that need throughput and not maximize the throughput device that do not need through ok, alright.

So, we looked at the state inside the CPU. Now what is the logic of a CPU, right? So, the way CPUs have evolved is basically that, CPUs have you know divided the logic of a computation into hardware and software, right. And so, hardware implements what are called instructions. And software implements the execute these instructions or sequences these instructions to implement it is logic, right.

(Refer Slide Time: 18:09)



So, the hardware designers are basically implementing instructions and the software designers writing these instructions. And the logic of a CPU, is basically an infinite loop; if I were to write it and see let us say is just you know an infinite loop forever run the next instruction.

So, you just say is what is the next instruction run it, what is the next instruction run it that is the logic of a CPU, right. And the software is basically a sequence of instructions that needs to be run and when it is put in the CPU with the CPU starts running those instructions.

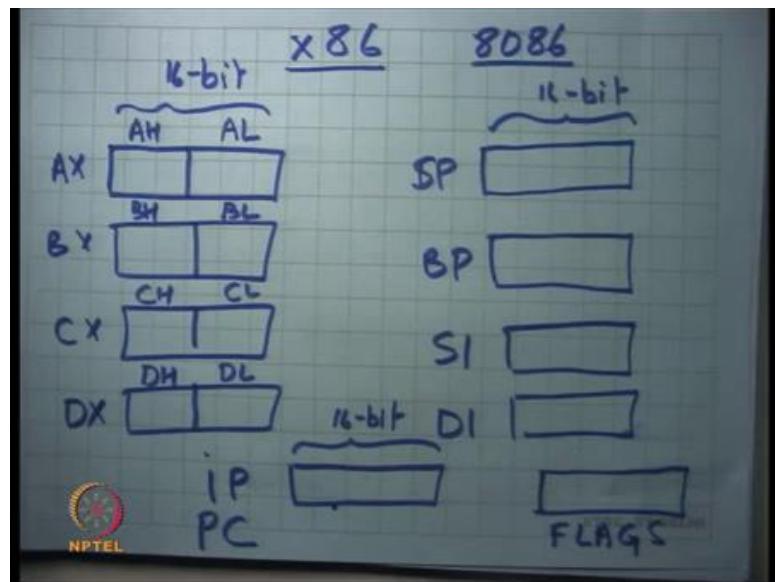
Student: Sir.

Yes.

Student: So, but this logic implemented through the logic gates and finite state machines on as in all like actual code is there for this.

So how are these instructions actually implemented in hardware; are they implemented using logic gates and finite state machines or is some other sort of software sitting inside the hardware that is actually emulating these instructions? Both are possibilities, there are time and space tradeoffs and in doing one or the other and both approaches have been tried. But let us for now assume that these instructions are implemented using logic gates and finite state machines, right alright ok.

(Refer Slide Time: 19:35)



So, now let us concentrate on this on the x86 architecture. So, x86 architecture has four data registers, they are called AX, BX, CX and DX; these are 16-bit in length in width, right. So, I am actually talking about the 8086 architecture, you know that was perhaps the first generation of these chips that we are using today, ok. So, you know let us say roughly in early 1980's or something.

And so, this architecture had you know, at that time computers were actually operating on 16-bit values and it was felt that operations on 16-bit values is enough for most purposes and so, these this chip was basically an a 16-bit chip. What does it mean for a chip to be a 16-bit chip? It basically means that, the registers width is 16-bits and the instructions that execute are also going to execute on 16-bit values, alright ok.

Also you know they say that most, the many instruction that only need to execute on 8-bit value; so they actually you know divided each resistor into two halves, two 8-bit halves and they also allowed instructions to name them separately. So, each of these 8-bit halves was called be AL, AH, BL, BH, CL, CH and DL, DH etcetera, right. So, an instruction could choose to run on the full 16-bit value or an instruction could actually choose to run on a 8-bit value; and the 8-bit value could be named using one of these identifiers DL, DH and so, you could do that, alright ok.

So, this was you know this was imagined to be primarily for integer computation, like you know whatever you want to do add subtract multiply etcetera. And then you know

you also want a wait; you also want some registers to be able to access memory, right. So, to generate addresses for memory and so, there were four more registers that were present in the x86 in the 8086 architecture. And you know they were called SP, BP, SI and DI, right these were also 16-bit.

So, the idea was that these registers could perhaps primarily be used for providing addresses to memory, right. And, you know the naming was also suggestive SP stands for stack pointer, BP stands for base pointer; I am going to discuss what a base pointer means really and SI and DI stands for I think source index and destination index, in any case these are four registers. And in fact, you know the way they design their instructions they could operate on any of these instruction registers equally.

Apart from this there was also another register which is called the instruction pointer, alright or let us call it you know it is also commonly known as the program counter. Instruction pointer or program counter same thing, this is also 16-bit, alright. This is also 16-bit, it is a register in the CPU and the semantics are that, when I showed you that loop that it is going to run the next instruction in an infinite loop.

How does that know what is the next instruction? It is just going to dereference, it is just going to send the value of IP; perhaps the value of IP is going to go through the memory management unit, the address is going to go to in the bus, the memory is going to serve the contents of that address and that is the instruction, that contents of that address is basically the instruction that gets executed.

So, in other words the program itself will live in memory. So, this was done for simplicity, they can be other schemes where the program lives somewhere else. So, you know there have been architectures that have separate memory for instruction then separate memory for data. And so, you know they optimize path for instruction, execution and they optimize the path for data execution separately, but the word, but here the same memory is holding both instructions and data, right.

Student: Sir base pointer is same as the frame pointer.

What is the base pointer? Let us just wait, right; now we are going to talk about a base pointer, alright. The other semantics of IP is that, it gets automatically incremented on every instruction execution. So, you execute an instruction, IP gets incremented

appropriately to point to the next instruction. So, that is also part of the logic of the CPU, so that you do not have to have a separate instruction software instruction; I mean that would be that would not be possible so, right. So, I mean to be able to do this for loop, you basically want that the semantics of the instruction pointer that it gets incremented on every fetch of the instruction, alright.

Student: Sir.

Yes question.

Student: Would a program want to access a data than memo address greater than 2^{16} that how many your program counter will address them

So, what if have you wanted to access an instruction that lives inside memory, which is greater than 2^{16} , right. In the scheme so far, we have discussed so far it does not seem possible right; because the address is only 16-bits and so, you cannot have more than 2^{16} bytes in memory, assuming a byte addressable memory.

So, let us for now just assume that the maximum amount of memory that you can have is 2^{16} that is it 2^{16} , so for gave it. But actually, it is not. So, you know 8086 actually allowed bigger memories and how it allowed, so you know discuss later. But basically, it is the MMU, the memory management unit that allows you to do greater physical memories, alright good.

And also the IP or the IP gets incremented on every instruction executed, on every in execution of an instruction; plus there are special instructions that actually can manipulate the behavior of IP, like the jump instruction right or the jump conditional instruction or the call instruction or an indirect jump instruction or a return instruction.

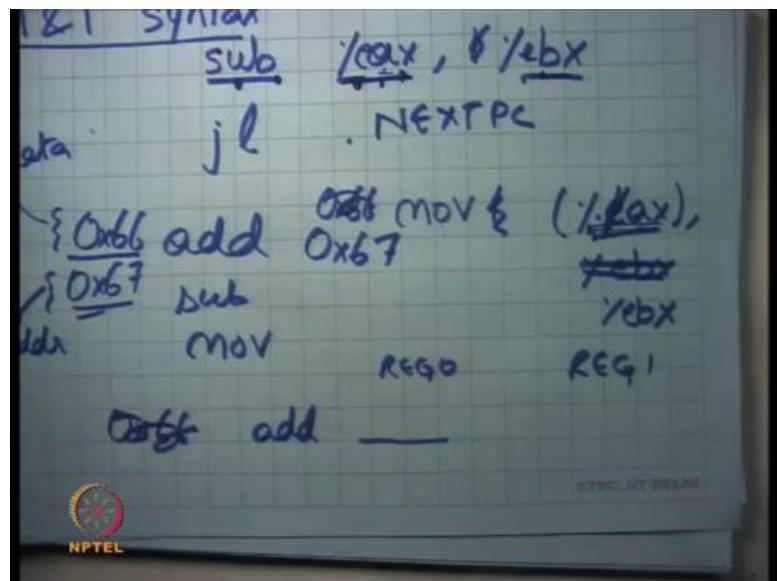
So, these are all you know different flavors of instructions that allow you to be able to manipulate the instruction pointer. There is no instruction on x86 that can actually just allow you to write to IP directly, right. If you want to write to IP you should, if you want to say write value x to IP; the way to do that is to have the instruction jump x ok. So, so IP is different from all the other registers in that sense, alright.

One more thing there is another register inside x86 which is called the flag register, alright. This is also a special register, you cannot just read or write to the flag register

like you can read or write to the other integer registers; but it stores information like whether the last information last arithmetic operation you performed overflowed or not, right.

So, there is some semantics for instructions. So, for example, if you said add AX and BX; if the result overflowed, then you know a flag in the and the flag register, one bit in the flag register will get set, alright. Similarly it can store information like whether the last computed result was positive or negative, whether it was 0 or not or whether there was a carry or a borrow in an add or subtract; it also has a system level information like whether interrupts are enabled or not, right. So, we are going to talk about that very soon.

(Refer Slide Time: 27:55)



And so, what happens is, the way you would actually do control flow typically is that; you would let us say in execute some kind of an arithmetic instruction, let us say you executed subtract and you say it subtract AX BX, alright. So, I am using some syntax here, this is the op code; you must have seen this in your computer architecture class. The percentage sign here means that I am talking about a register and whatever follows the percentage sign is the name of the register.

So, I am saying you know, the AX register and the BX register and we are using a syntax called the AT&T syntax for x86, right. There are the two types of its syntax of for x86 what we are going to use in this class and programming assignment is the AT&T syntax. There is another syntax called the Intel syntax, if you read the Intel reference manuals

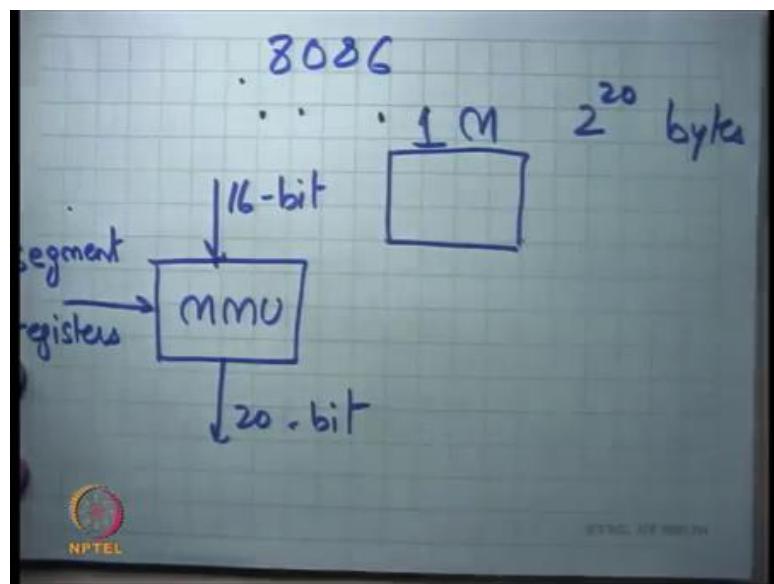
you going to find the Intel syntax; but this is this is you know this is more common and this is what we are going to use in this course, alright.

So, this the syntax says that every register needs to be prefixed with the percentage instruction. And, also says that the first argument is the destination, the first operand is the destination operand and the second operand is the source operand. And so, this instruction means, and they are only two operands you know for an instruction for an op code typically. And, in this basically means that replace the value in AX with AX minus BX right ok.

And, now if I want to say you know subtract AX from subtract BX from AX and then if the result is greater than 0 then jump somewhere, right. So, the way it will happen is that this instruction is going to set a bit in the flags register. And then there is an instruction which says let us say jump less than; which basically says if the neck if the negative bit in the flag resistor is set, so if the last operand was if the last computed value was negative, then you jump to some you know some other next PC, right.

So, these conditional jump instructions are a way to read the flags register and these arithmetic instructions are away to write these flags registers. A CPU a software cannot just directly read or write the flags register, these are the only two ways to read and write in those flag register ok, alright. So, you know there are other instructions like add, subtract, move etcetera that do the usual thing of their name suggests, alright ok

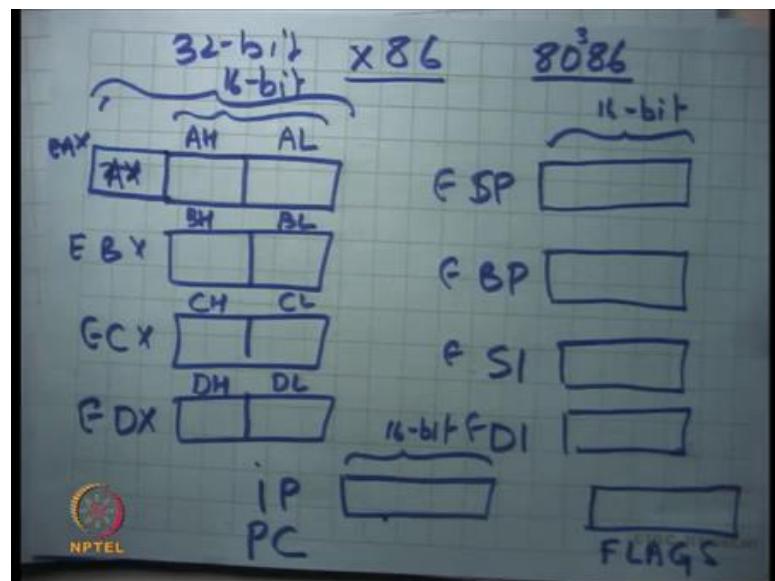
(Refer Slide Time: 30:33)



So, now let us come to the question that you know, do 8086 process, did 8086 processes support only 64 KB memory? So, answer is no, actually they supported up to 1-megabyte memory, right, so 1 M, right; that is the 2 to the power 20 bytes, right. And so, the CPU architects needed a way to be able to address this physical memory using 2 to the power 20 bytes and you know the architecture itself is 16-bit.

So, what they did was, you know the MMU would take in a 16-bit value and generate a 20-bit value with the help of certain register which are called the segment registers. Once again, I am talking about a particular architecture that is 8086, right ok. So, how does this work?

(Refer Slide Time: 31:29)



Apart from all the registers that we have talked about there are also, so let us say you know there is A B C D these are the integer registers, there is IP and there are flags and you know there are also those ESP etcetera, you know want to write them. And then there are these segment registers that, there are six segment registers in on x86 CS code segment, DS data segment, SS stack segment and then there are and then there was another segment which is called the ES, this is you know for anything else you want to do it is typically used for strings.

So, it is called you know, it is usually they are just used for the string operations and we are going to talk about that implement. These registers themselves were 16-bits, right and what MMU would do is; so, any at if you want to dereference memory, you cannot just

say that here is a register and I want to dereference the address at SP. You have to actually address the memory using a pair of the segment register and an address; and the address could be a direct address or a register address or whatever else, right.

So, I would either say you know %CS: I %IP, this is the full address that I want to access. And the semantics of this is that, the value of CS, so when I put something in brackets it basically means; you dereference that, right. So, actually do not dereference it here. So, you the value of CS that is %CS into 16 plus the value of IP; so, that will be your total address, that will be your complete address that will actually go on the bus to the memory, right.

So, in other words your MMU on 8086 was this very simple MMU the memory management unit, which would take an address named by a pair of segment registers and some integer which could be used; which could be the resistor, one of the resistors integer resistors and it will do this computation and send the address on the physical bus right.

Student: Sir, 16 or 2 to the power 16.

16.

Student: Only 16.

Yeah. So, basically it only allowed up to 1 megabyte of memory that is 2 to the power 20 bits. So, 16 is 2 to the power 4. So, you can imagine you can see that, this totally this can address any byte in a space of 2 to the power 20, it 2 to the power 16 into 2 to the power 4 is 2 to the power 20, so you can address any of the 2 to power 20 bytes ok, alright. And plus, you know there was some default things.

So, for example, any dereference through the IP will always go through the CS, right. So, you know for example, when I say run next instruction, the run next instruction is only not only dereferenced in the IP, it is dereferencing CS: IP, right. So, whatever you have set up in CS that is going to you be used in your MMU. So, another the intention must have been that, you know the code will live in a separate segment and you are going to or separate region of memory. It is all uniform memory; but you are basically dividing this memory artificially, using these segment resistors.

And you will basically say that CS holds the base of the area where you going to store your program, DS holds the base of the area which holds your data, SS holds the base of the program which holds your stack, right. And then also they were default things like, whenever you dereference an instruction it is going to go through CS; whenever you dereference the stack using the SP or BP pointers, then you are going to go through the SS segment. And everything else by default is going to go through DS right and then there was some special instructions that will go through ES.

Student: Sir certain, but you think CS has 16-bit resistors is not it page full of memory, because we can just you used as the 4-bit register and still perform the same work.

Could we have used CS as a 4-bit registers and perform the same operation; well a 16-bit register allows you to you know start anywhere, right. So, the code segment could start on the top of the one megabyte region; but if you have a 4-bit register, then you know you have you lose the you lose some freedom in where the base will start, ok. So, and you know and from a hardware designer's point of view all my logic is anyway 16-bit, you know having such special 4-bit where all logic is actually more costly than less costly. So, I am just guessing, yeah.

Student: Is there any specific reason of keeping memory 1 M.

Is there any specific reason, no; I mean at that time they just thought you know 1 M is more than enough. So, let us just. So, every memory will have a limit, even today's memory you cannot, I mean the CPU will only support of memory up to this size, right. And so, whenever you know in the real world you will always have limits, you cannot just have infinite memory and at that time 1 M was thought to be enough and of course, we know now that is not enough.

And so, the architecture has evolved, and I am really going to talk about how it is evolved, it is basically you know it is evolved in a backward compatible way, alright. So, I mean this was a little you know I mean not very clean actually, because there is no protection and I can you know, I can reference the same byte in physical memory using many different addresses, right. I could set up CS in some way and IP in some way or CS something CS lower IP higher, because the same byte could have multiple addresses in this scheme.

Because as you pointed out there is redundancy and in the number of bits are being used to address a certain address space right. 32-bits have been used to address subspace of 2^{20} , so there is clearly redundancy and the way things are done, alright. So, it is a little complex, it is it was also a little difficult for to program this kind of a thing; because you know we are used to writing programs where we talk about variables and now it is a job of a compiler to actually translate those variables into memory addresses.

And now the compiler has to worry about where this memory address is going to be allocated, whether it is DS or SS or CS. And also, it has to worry about oh there should not be any overlap, so you know CS plus some address should never be able to reach this variable you know. So, all these things make life very difficult for the software writer or in particular the compiler writer if you are using a compiler, alright.

So, clearly 16-bits was painfully small, quickly this was realized you know in a few years time that you know this is not going to work we need to extend this; but you know what they also needed to do was provide backward compatibility. So, there is some software that is been written for the 16-bit architecture, you know there has been OS, there are some OS is that have been written for the 16-bit architecture.

And they have you know they have invested in the company, they have bought the processor, they written software for that come for that processor and now if the processor gets changed then you know they will cry. So, what they do is they provide backward compatibility. And the backward compatibility means that; when the process starts, the processors boots when you are power to on a processor, it actually always boots starts in the 16-bit mode.

So, if you know OS for written for 16-bit, it will just you know happily run on the 16-bit mode. And then you provide certain special instructions, which were actually not present in the 16-bit processor to switch to 32-bit mode, right. So, if an OS is actually a 32-bit OS, it will start in the 16-bit mode and very soon it will call an instruction to switch to 32-bit mode and now it can run in 32-bit mode. However, if there was an OS which was which does not understand 32-bit only understand some 16-bit, it can still happily run on the old processor, right.

And actually this has continued and to this day you know when we are when we have our chips which are 64 bit, we still I mean boots in the 16-bit and it is a job of the OS writer

to you know, the OS writer is aware that it will always boot in the 16-bit. And now it will execute some code in the 16-bit mode, call us a special instruction to switch to 32-bit mode; then it will execute some code in the 32-bit mode, then we will call a special instruction to switch to the 64 bit mode and that is you know finally, I can run my applications, right.

So, the architecture developed from 16-bit to 32-bit to 64 bit and to provide backward compatibility it will still boots in 16-bit and then third and the OS has to switch it from 16 to 32 to 64 in that way, right. So, that older OS is can still run. So, for example, the 32-bit OS can still run on a 64-bit processor and the reason is basically because it is still you know boots in the 16-bit mode and then 32-bit, alright ok. So, how did the 32-bit architecture change?

So, the 32-bit series was let us say you know 80 something 386, 186, 286 etcetera and they basically made all these registers 32-bit. So, you know this was extended by another 16-bits, this becomes 32-bit and the new name of this register is EAX, extended AX, right.

Similarly, this becomes EBX, ECX, EDX, ESP, EBP, ESI, EDI, alright. So, all these registers get extended by an extra 16-bits and they are renamed to ESP, EBP etcetera. They still allow you to access the lower 16-bits of a register by using the old names. So, you can still say AX and that will mean the lower 16-bits of EAX, right or you can say SP and that will mean the lower 16-bits of ESP, right ok.

And now they changed all these instructions to you know with the same op codes to mean the same thing except that they will now operate on 32-bit values, alright. So, subtract basically means subtract EAX, EBX alright; add, move whatever right, they all basically become 32-bit, they become 32-bit versions on themselves, right.

So, the same op code, the same encoding of an instruction becomes a 32-bit version of itself. If for some reason you want to still access a 16-bit region, a 16-bit value inside the register; you can prefix an instruction with our special prefix. Let us say the prefix is you know some bytes let us say 0x66, it basically says that treat this instruction as a 16-bit instruction as opposed to a 32-bit instruction.

So, these are just some things that the designers did. So, to be able to maintain both worlds, you can know by default everything becomes 32-bit; but if you want to use 16-bit you can prefix an instruction with this special prefix and that instruction now behaves as though it is a 16-bit instruction, alright. And similar things are actually true about 32-bits and 64 bits ok.

Student: Sir.

Yes

Student: Do not we prefix with this 16-bit, the prefix to make a 16-bit then the opponents have to be the older ones that is the AX, BX not EAX, EBX.

Right. So, for example, if I say 0x66 add and let us say you know I specified no. So, there is some number encoding for each register right; for example, AX the encoding for AX is 0 and the encoding for BX is let us say or say let us say 1 hypothetically, right. So, it will say you know add register 0 to resistor 1, right. And if I do not use the prefix, this encoding means add EAX to EBX; if you use the prefix it basically means add AX to BX, right that is that only difference.

And they are actually two prefixes there is 66 and 067 which stand for you know, this basically says I want to change the size of the data. So, data is basically the value that you are operating on for example, AX or EAX and 67 basically means I want to change the value of the address, right.

So, for example, if I wanted to dereference memory, I could actually do something like move percentage EAX to percentage EBX. Oh! did I say that the first operand to the destination sorry, so the second operand is the destination, alright. So, in this in that in the syntax of AT&T the second operand or the last operand is the destination correction, alright.

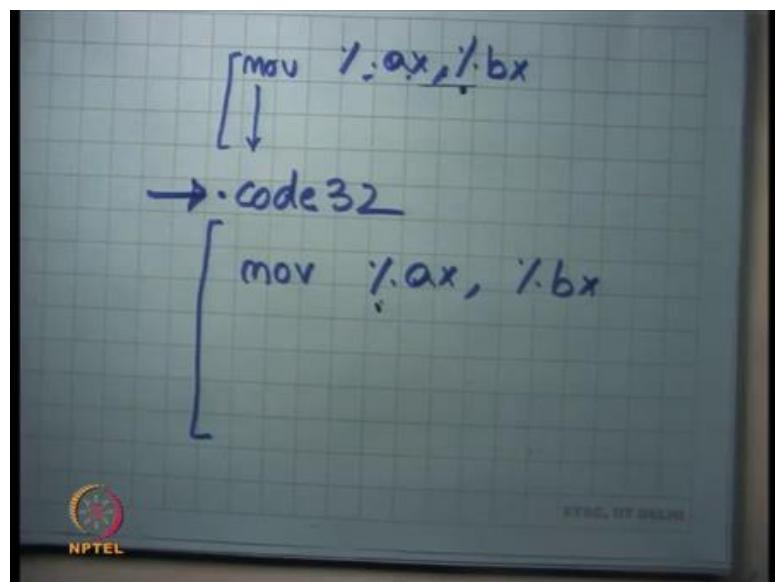
So, when I say move EAX to EBX, it is basically going to read the value in EX, dereference it; which means it is going to you know send that address to the MMU, generate a physical address, ask them memory what to fetch it is contents and the contents are going to be stored in this register called EBX. So, that is what is going to happen, right.

If I prefix it with 66; what is going to happen is, the address remains 32-bit, but the data becomes 16-bit, right. So, I still use 32-bits to dereference memory to address memory through my MMU; but the number of bytes I fetch from that address is only 2 or 16-bits, right. So, that is what the prefix 66 means, it is basically referring to the size of the data it is.

So, if you see you if you prefix it with 0x66 let us say, then you are going to you basically changing toggling the size of the data on which this instruction is operating. On the other hand, if I use 67, I am toggling the size of the address, right. In this case the address is EAX. So, now, this instruction is going to become move EAX and if I do not have 66, then it is going to say move AX to EBX. So, it is basically saying dereference using AX, but the data still remains 32-bit right and if I use both then both becomes 16-bit, right.

So, they are two things does in every instruction will have an address component to it and a data component to it if it accesses memory. And so, these two prefixes can toggle either or both, alright yeah. So, lot of details, but good to understand good to know once and then you know be comfortable order it forever, alright.

(Refer Slide Time: 46:39)



Let us also understand you know how an assembler works. So, for example, when you write code, when an OS is going to write code is going to write some assembly instruction. So, it is going to say move something, something and so on. So, you can tell

the assembler that look this part of the code is, compile this code as 16-bit code and then you know you can tell the assembler.

So, there is a directive called let us say code 32 and then you can tell that all future code should be compiled as 32-bit code, right. So, for example, if there is an instruction called move AX BX here and there is another instruction called move AX BX here; here I could just use the encoding of this instruction without having to prefix it with anything, right. But here if I want to use the same instruction, I need to prefix it with 0x66 let us say right, because it is a 32-bit mode.

So, this directive is a way to tell the assembler that in what mode should you compile this code because, the same string can mean different encodings in different mode ok, alright. So, will we look at 32-bit architectures for our programming assignments and most of our discussion; but you know seems already seems very complex I do not want to discuss 32-bit or 64 and I do not think you want it discuss it either, right. But you can imagine it is a similar kind of complexity that is going to occur and in doing that also, alright ok.

(Refer Slide Time: 48:07)

AT&T Syntax	"C"-ish equivalent
movl %eax, %edx	edx = eax
movl \$123, %edx	edx = 0x123
movl 0x123, %edx	edx = *(int*) 0x123;
movl (%ebx), %edx	edx = *(int32_t*) ebx
movl 4(%ebx), %edx	edx = *(int32_t*)(ebx+4)

Let us look at some instructions, some real instructions. So, here is the AT&T syntax. Let us say move 1 percentage EAX to percentage EDX. Let me write the C-it equivalent or you know semantics of this instruction in a language we understand C, right. So, let us say.

So, what this basically means is, EDX is equal to EAX right just to understand what this means; if you do not understand the syntax let me just explain it to you and see. I could say move \$123 to EDX; \$123 means it is a constant; it is an immediate value, right. It is an immediate operand; you must have seen immediate operands in your computer architecture class, and this basically means EDX is equal to 0x123 let us just say it was you know it was a hexadecimal number 123.

Notice the use of the dollar sign to specifically say that this is a constant that I want to move into EDX. On the other hand, if I say move, so, I am also prefixing it with a character which says what is the size of the data I am using. So, by default on a 32-bit mode it is 32-bit, but you can also specify it using l. So, l stands for 32-bit. So, you want you will basically be moving 32-bit values, alright.

You can also say move l's 0x123 to percentage EDX. Notice the difference between this and the previous one is that, I am not using the dollar sign, alright. And what does this mean? Yeah so, treat 123 as an address for the memory, right. So, it is basically saying EDX is equal to dereference; first typecast 123 as a pointer, that points to a 32-bit value because the 32-bit instruction and then dereference that pointer and whatever result you get put it in EDX right; just dereferencing EDX and dereferencing 123 ok.

If I say mov l, so this is immediate addressing, I could also say move l EBX to EDX; this is indirect addressing basically means EDX is equal to star you know it is int32 just to specify that it is a 32-bit number (int32 *)EBX, alright. So, basically saying look at the value in EBX, use it an address to memory, dereference it in memory and fetch the contents and store it in EBX, EDX ok. And finally, I could you also say something like movl 4(%EBX) to %EDX.

So, x86 allows you these kinds of instructions where you can specify an offset with a resistor. So, basically means that you add 4 to EDX and then you dereference it and then you get the contents and put it in EDX. So, basically means EDX=*(int 32 *) (EBX + 4). What is the difference between the prefix 0x66, 0x67 and this suffix l in my assembly code, this is assembly code, alright?

So, I can use you know, I am specifying assembly code using strings where I specify an op code using a string and I use a as a suffix l to indicate whether the 32-bit over 64-bit. The 0x66, 0x67 is for the instruction encoding in binary, right. So, the assembler is going

to convert this string into prefixed or un-prefixed versions of instruction encodings, right ok.

So, I will stop here.

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

Lecture - 05
X86 Instruction Set, GCC Calling Conventions

Welcome to Operating Systems Lecture 5. Yesterday we were we started looking at the hardware interface for an operating system and we started with the x86 architecture. And, we were looking at the instructions set of the x86 architecture just to get ourselves familiar with all the instructions.

(Refer Slide Time: 00:25)

AT&T Syntax	
movl %eax, %edx	edx = eax register mode
movl \$0x123, %edx	edx = 0x123 immediate
movl 0x123, %edx	edx = *(int32 *)0x123; direct
movl (%ebx), %edx	edx = *(int32 *)ebx; indirect
movl 4(%ebx), %edx	edx = *(int32 *)(ebx+4) displaced
movw %ax, %bx	

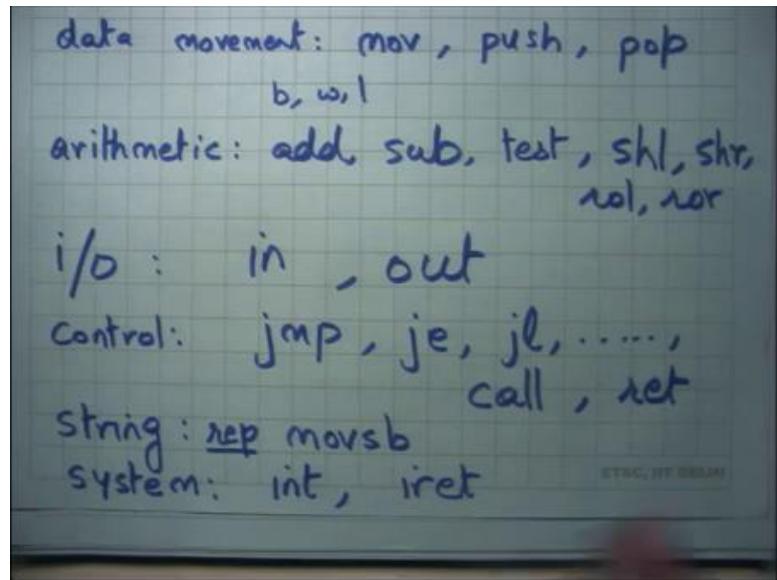
And you know we said that the so, we here the syntax of the x86 instructions on this side and this is the AT&T syntax where the destination operand is the last operand and the source operand is the first operand or you know if the three operands in the first and second operands are the source operands.

And, you know there are these different addressing modes which you may have already studied in your computer architecture class. So, you can move register to register. This is a register mode. So, every instruction can operate in either these modes. We know we could very well replace move with add or subtract; that means, the same thing you know it will it means a corresponding thing mov 1 dollar 0x123, this is the immediate mode. Basically, means that assign immediate value one two three to this register.

Move without a dollar basically means treat this address as a memory address and so, this is the direct memory addressing mode right; you may have seen it in your computer architecture class. Then you could say you know if you bracket the register name like this, then this basically means it is an indirect addressing mode basically means use the address in EBX to dereference memory and get the contents into the destination operand right.

So, in the C equivalent it basically means $EDX = *EBX$ where EBX is treated as a 32-bit as a pointed to a 32-bit value. And, then there is also the displaced mode. So, you can specify a displacement along with the indirect mode. So, which basically means that add 4 to EBX and then dereference it and these displacements or immediate values or memory addresses can be 32-bit byte also in 32-bit mode on in x86 right. So, you could have a full 32-bit displacement with a 32-bit register value add them and then dereference it alright.

(Refer Slide Time: 02:37)



So, the different types of instructions on x86. There are let us say data movement instructions like move, push, pop right. We have already seen move push and pop are instructions that operate, specifically on the stack. I am going to discuss these very soon. Each of these opcodes can be suffixed with a character which indicates what the size of the operand.

For example, move l basically means the 32-bit operand and move b mean means an 8-bit operand and move w means a 16-bit operand. So, b, w, l is the I mean just syntax of AT&T to specify 8, 16 and 32 then there are arithmetic instructions like add subtract.

Student: Sir what are b, w, l can you explain?

What are b, w, l?

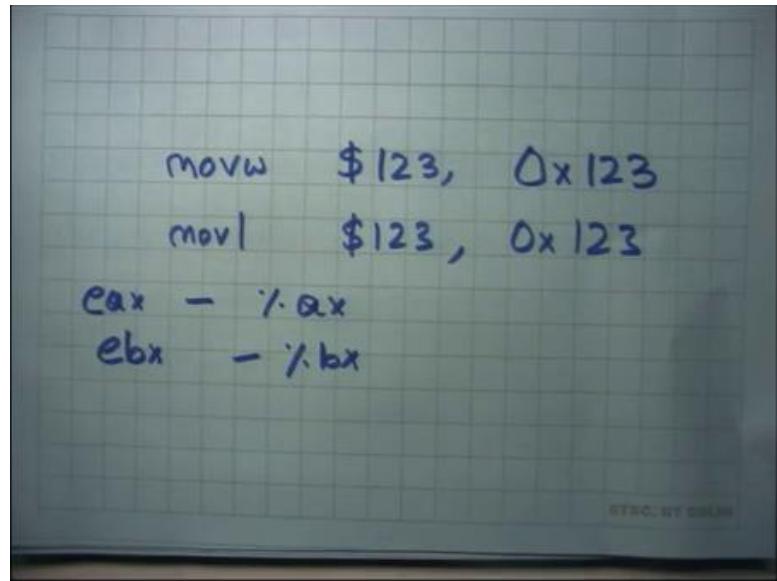
Student: b, w, l.

b, w, l are just suffixes to op codes which specify the size of the operand right. So, for example, in this here when I say mov I actually suffixing it with l, it basically means treat these values as 32-bit values right. If I add instead used b here, it would have meant I am I am working on 8-bit values right.

Actually, if I say b here this is an invalid instruction, this is an illegal instruction because you know I am specifying 32-bit registers and I am using b here. So, it is you know there are some instruction that are not possible. So, l is the only valid legal value here. But if I want to say for example, mov w I would want to say mov w ax bx right. Often the suffix is inferable from the operands.

So, for example, if I did not say w and I just said move ax bx the assembler would be able to figure out that it is a 16-bit operand. So, it you know it will it itself for example, suffix it with w, but for some cases it may not be possible to infer the operand size. For example, if I have something like let us say if I have something like mov \$123 to memory address 123 right. So, it is basically saying move this immediate value to this memory address right and there is ambiguity here because we have not specified the operand size and it is not possible to infer the operand size from the operands right.

(Refer Slide Time: 05:11)



So, mov w will mean treat this as a 16-bit value and store only 2 bytes to this address and mov l would mean treat this as a 32-bit value and move it to. So, in such instructions you are required to specify a suffix otherwise assembler will say there is some ambiguity in your code right.

Student: So, how do we address the first 16-bits of a 32-bit register?

How do you address the first 16-bits of a 32-bit register? Well x86 gives you this ability to say you know; so, the first 16-bit is of EX and AX right and the EBX are BX and so on.

Student: What about the last 16-bit?

You cannot act as a top 16-bit is of a register directly not allowed alright. So, add subtract you know then and we also said that these instructions also set up the E flags register appropriately. So, example if it is a negative value etcetera. So, they also write the flags and there are instructions like jump equal, jump not equal, jump less which are which read the flags to make control flow decisions.

Then there also instructions like test which actually do not do anything they just set the flags. So, test is going to check if a value is equal to 0 or not or things like that or you know you can also have things like shift left or shift right or rotate left or rotate right and

so on you know. So, there is a huge list of operands that can happen just to give you a flavor of what.

Then there are IO instructions like in and out right. So, so far what we have seen are basically instructions that are do arithmetic or data movement that operate on either registers or memory, but what if I wanted to access a device? There are special instructions which are called IO instructions and the instructions are IN and OUT and what they mean and what their syntax is we are going to talk in a few minutes.

Then there are control flow instructions like jump you know jump equal, jump less than, jump greater than and so on. So, lot lots of different things and then there are special instructions for function calls. So, there is an instruction called call and there is an instruction called return which are basically equivalent to saying function call and return and what their semantics are we also going to discuss in a few minutes alright ok.

And then there are some instructions which are called string instructions right. So, string instructions are of this type mov sb where sb stands for move a byte of a string right and the default and this instruction has default operands the memory address of the. So, it is a string instruction which says move 1 byte of data from the memory pointed to by register ESI to the memory pointed to by EDI alright.

So, this is a special instruction that can actually do two memory accesses in one instruction. It will read a byte from the address of ESI and write a byte to the address of EDI right. And the reason this is supported in the x86 architecture is because there are many string operations that are usually done and you can actually prefix this instruction with what is called a repeat prefix that basically executes this instruction repeatedly each time incrementing ESI and EDI.

So, ESI is considered the source address or the source pointer, EDI is considered a destination pointer, mov sb means move a byte from the source pointed to the destination pointer prefixing it with repeat basically means keep repeating this. And, each time keep incrementing ESI and EDI till a certain condition has reached and you can also specify what under what condition you want to stop.

So, you can imagine you know the hardware designers what they are had in mind was let us say things like string copy right. So, if you want to implement string copy one way to

implement, it is to you know implement in using many instructions like a for loop which basically you know increments a counter in software. But because it is such a common operation you know the hardware designer said let us have a special instruction for it. And, there is just one instruction which will for example, do mem copy or string copy or things like that ok. Just to know because I mean just in case you see this kind of code in your programming assignments you should know what it means alright.

And then there are you know special instructions which we can call system instructions. These instructions are very, are instructions that are required by an operating system. So, so far, the instructions we are really talked about are instructions that user program may require accessing memory, accessing registers; accessing IO may or may not be allowed for a user for an application.

So, even in and out can be considered system construction in that sense, but then there are special instructions for system which you know allow you to for example, raise an interrupt right. What does means is it is going to it is called a software interrupt instruction which basically emulates that I have received an interrupt.

how that is used what it means etcetera we are going to look at in a moment, but just to give get give you a flavor of what exists and then there is another instruction called iret which says return from an interrupt right. So, you can say you can simulate an interrupt you can get into the interrupt handler and then you can have a return from interrupt instruction ok. Never mind, we will be going to look at it in more detail in a few minutes.

Student: Sir in and out instructions are like do they take input from keyboard and tend to console or?

So, what do they you know in and out instructions mean? Just hold on we are going to discuss this very soon right.

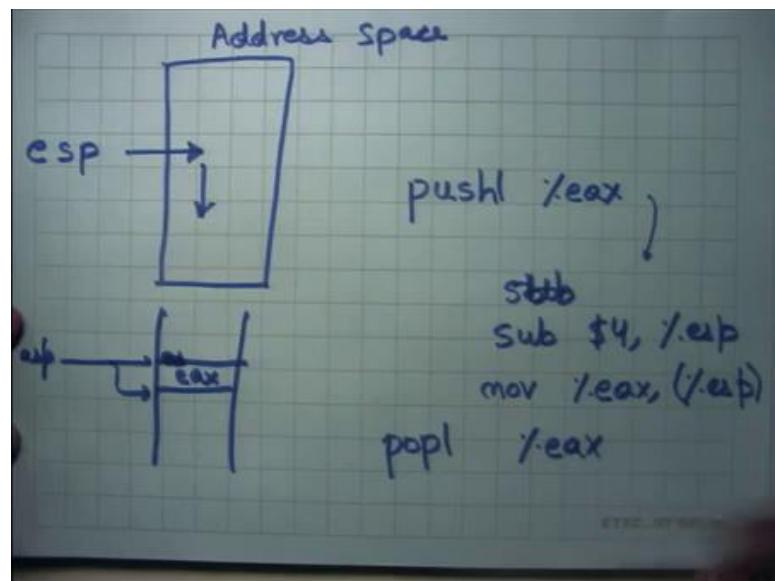
Student: What is test?

Test is an arithmetic instruction that just tests let us say your operands to see if they are equal to 0 or not or they at ands them and sees what. So, I mean it is an example of an instruction that does not necessarily modify it is operands, but set some flags based on the values of the operands. So, often you know if you want to execute control flow, you

can execute test and then you can execute jump conditional to or based on the value of the test alright ok.

So, we are going to discuss in a you know in the next couple of lectures how system calls work how, how operating system instructions work etcetera, but before that let us just talk about how function calls work right. So, how do you when you call a function what happens what kind of code is generated by the compiler and what kind of you know. So, how does it all work inside? It just in the context of function calls before we start understanding how things work for a system call and how does OS do thing different things etcetera alright.

(Refer Slide Time: 12:23)



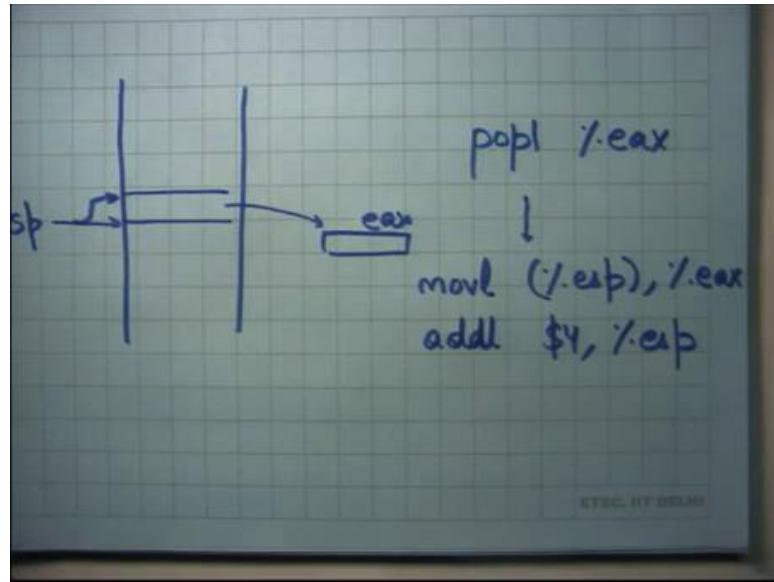
So, every process has an address space alright and in the address space, there is one pointer which is the stack pointer, or you know the 32-bit x86 architecture it calls with the esp extended stack pointer right. And this pointer basically the semantics of this pointer are that it should it points to the top of the stack and. And so, and the other convention on x86 is that that stack grows downwards right.

So, the stack is initialized at some value and then you know if you for example, call a function, then the stack is going to grow downwards; if you allocate a local variable the stack is going to grow downwards, assume we have seen the use of stack in other courses like programming languages or architecture for doing function calls right yes alright.

So, there are some instructions that actually access the stacks. So, for example, push I can push a register onto the stack, or I can push an immediate value, or I can even push a memory operand onto the register right. So, this can be replaced by a register immediate or memory whatever you like. Basically, means the semantics of this are subtract 4. So, in this case it is a push 1 because I am pushing a 32-bit value; 32-bit means 4 bytes. So, it basically means subtract 4 from ESP grow the stack downwards and mov EAX to the memory location pointed to by ESP right that is what push means.

So, here is an ESP you call push, it is going to decrement stack and it is going to put EAX in that. So, in other words what is going to happen is if this was ESP, let us say this was ESP and it is pointing here then when you call push what will happen is ESP is going to now point here and this memory location will now contain the contents of EAX right ok. Similarly, pop EAX basically has the opposite semantics.

(Refer Slide Time: 15:09)

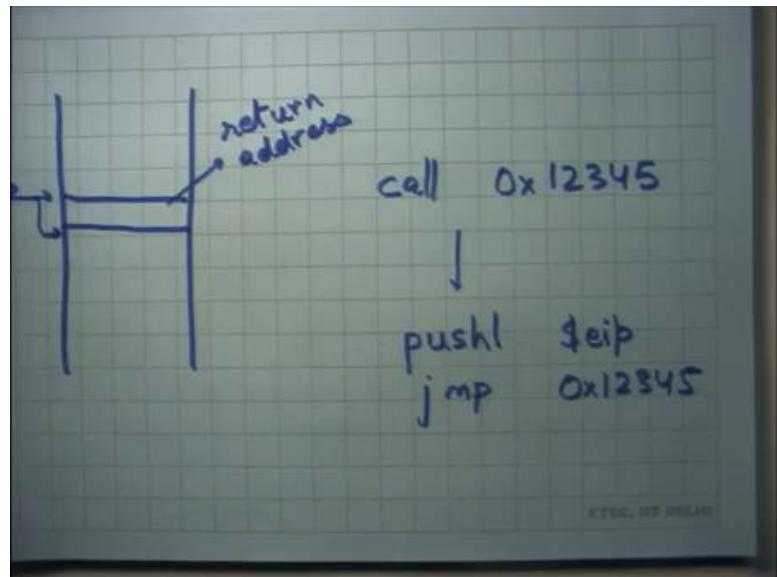


So, let us say this is ESP, I call pop EAX what it is going to do is it is going to move ESP contents of ESP into EAX and add 4 to ESP alright. So, in other words let us say ESP was pointing here, you executed pop what is going to happen is whatever was contained here is now going to go to the EAX register and stack pointer is now going to point here right. So, push grows the stack pop shrinks the stack alright.

Then there is function call right. So, we looked at we saw that there is an instruction called call. So, I can actually say call you know some address 1 2 3 4 5 which basically

simulates a function call. And so, in this case what happens is let us say here is the stack and before this call instruction ESP was pointing here.

(Refer Slide Time: 16:28)



So, call instruction is going to do what? It is going to push the current EIP right. Notice that no such instruction exists; I am just using it to explain it to you what the operations of this are right.

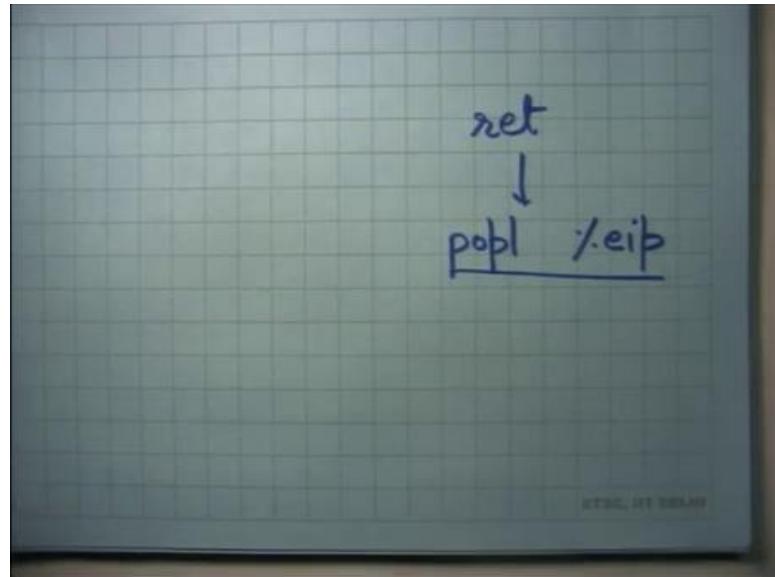
So, it is going to push the current EIP after it has been incremented. So, you know the as soon as instruction is fetched, the EIP gets incremented and the incremented EIP is what you push onto the stack right and then you jump to alright. So, basically you just push the address of the next instruction on to the stack and you jump to the destination right. So, in other words what happens is ESP gets decremented stack grows and this contains the address of the.

Student: Call.

Instruction following the call site; the instruction following the call site right; this is also called the return address right. You can imagine why you need the return address on stack because when the function calls return that is where he is going to jump back right. So, that is what it means right when you make a function call it is going to execute that and then when it calls return you are going to come back to the next instruction. So, that

is where the way it is done is basically using the stack and a return address stored on the stack alright. And then there is the return instruction.

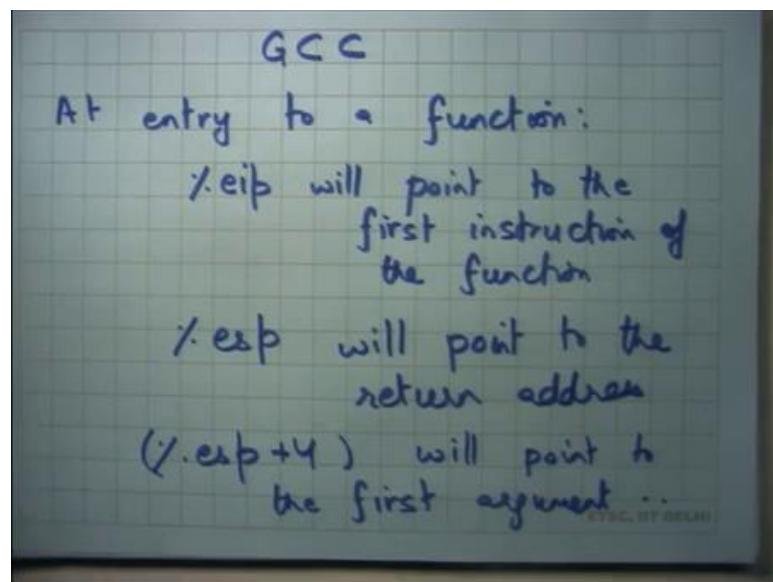
(Refer Slide Time: 18:19)



Basically, what it does is it just pops from the stack and puts it in EIP right. Once again this is not a real instruction. I am just, I am just using it to explain what return does right. So, it will increment the stack pointer which means it will shrink the stack and whatever the value was there it is going to put it in the program counter or the EIP and so, that is where you go next right. So, it is an indirect branch in some sense alright.

So, a compiler usually dictates how the stack is going to be used right. So, a compiler will have some conventions on how the stack will be used in case of a function call right. So, typically you would have different functions in different files and each function will be compiled separately and the compiler will have some conventions on how this function body should be organized such that it can fit in well with the caller and its callees right. So, let us look at what kind of what kind of conventions does a compiler have.

(Refer Slide Time: 19:28)



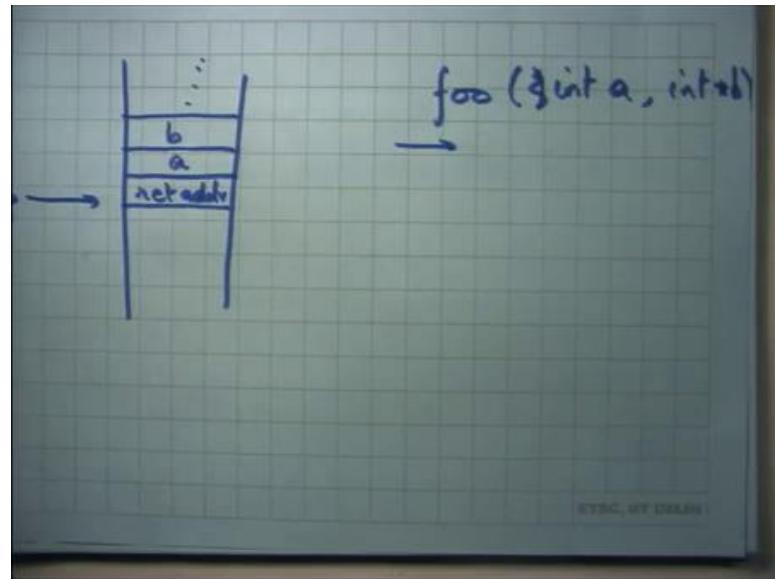
And let us look at one compiler which is which is a popular compiler that we all use is GCC alright. So, GCC says that at entry to a function EIP will point to the first instruction or the function right. Obviously, when I call say call some function name, the next EIP the next program counter should be the first instruction of that function right, ESP will point to the return address right. So, we saw when you say call the return address automatically gets pushed and so, if you execute call you know your GCC is basically just using the same thing as what the call instruction does right.

So, if you are inside a function call as soon as you enter the function call the program counter is the first instruction of the function and the stack pointer should be pointing to the return address right. So, if the first if the first instruction of the function is let us say return, then you just go back to the return address alright. And, ESP + 4 alright so, this is as you can see it is a very 32-bit architecture specific thing 4 bytes for the return address. Let us assuming that return address is 4 bytes and ESP + 4 should point to what.

Student: Unit is.

Yeah, the first argument of the function alright.

(Refer Slide Time: 21:42)



So, in other words if there is a function foo and I make a call to foo at this point; my stack layout must and let us say the foo takes you know two arguments int a int star b then if at entry to the function ESP should point somewhere in the address space wherever it points this should be the return address right and this should be a right. And, this should be b and so on right. If there are multiple arguments, they just stack on top of one another right.

So, that is the convention of a compiler. So, that is you know that it uses and that is what most compilers will use that a function when it starts. You can assume the following things about the stack layout, it can assume that the return address that ESP and the first argument is that $ESP + 4$ and all the flags and arguments are above it right. So, it is a responsibility of the caller to set up the stack in such a way before it calls the function.

Student: Sir.

Yes.

Student: Why is it sir when the function after the function call it will use them in the opposite way right? We can use a and b and then the return address. So, would it be would it be easy if we study opposite way that the return address is above?

Interesting question. So, why is return address? So, this is in some sense in the top of the stack right. So, why is the return address at the top of the stack and why are the arguments below it, why cannot it be the opposite way?

Student: Because push in the end.

Yeah so, firstly, the number of arguments is variable. So, you know it is hard for it is much easier to say that return address is right at the top because otherwise you have to worry about how many arguments there are and based on that you have to compute whether return address is going to live. So, as opposed to that because return address is the most common thing that is going to happen and so, let us have it at the top.

Secondly when you are calling the function you know that is when you want to push the return address. And so, you are going to push something and then you are going to call the function and that is when you want to push the return address because after that is the next instruction that needs to be executed. Otherwise you know I would have to worry about where I have, where I should return, and I have to push that explicitly ok. So, the instruction that pushes that call that actually makes the call and pushes the return address should be the last instruction in the caller.

Student: So, when we want to actually use a and b that we have to go up you can keep return address in your register using b, then bring it back.

So, if I have to use a b, then I have to actually go up is that right.

Student: As a just like a RAM.

It just it is just RAM right. So, it is I do not have to go up I just specify an address I just say $4 + ESP$. So, it is actually no less efficient than say ESP right even I can say a 100 plus ESP . It is equal in efficiency; it is just an address computation alright

Student: Sir.

Yes

Student: Even x86, there is no function of link register that is $l r$ which is in arm assembly language has $l r$ which is link register.

Student: So, does it have.

So, so, the question is really about arm and x86 arm has a link register and x86 does not have a link register that is true why and why not let us just let us defer that discussion. But x86 does not have a special register where it stores the return address x86 in state stores the return address on stack right. Other architectures like arm have a special register where they will store the return address and now it is the responsibility of the software to actually move that a register to the stack in case of nested function calls right. So, that that overhead or that extra work from the software is avoided in x86.

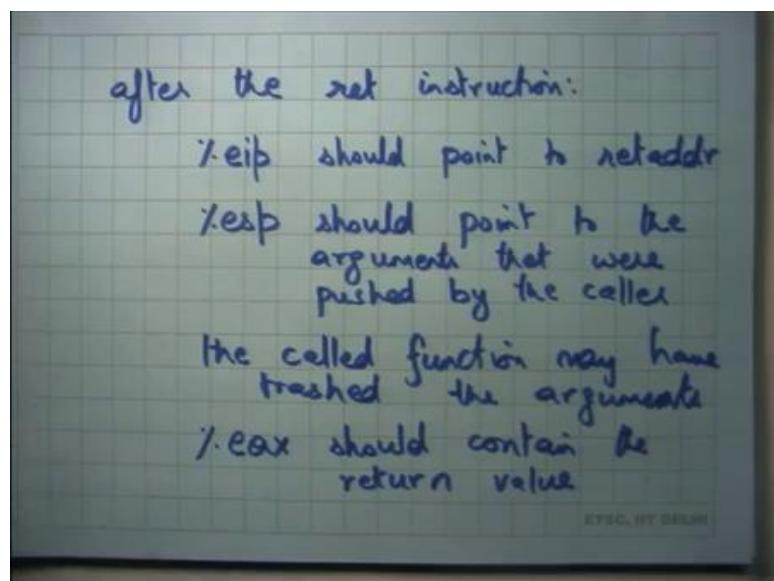
Student: Sir, we use branch and link and automatically copy that to.

Sure, if you use branch and link what it does is it sets the return address into 1 r that link register right. And now if you make another function call, nested function call then you before that you have to save the 1 r onto the stack.

Student: Yeah.

So, x86 avoids that in some sense alright and. So, that is the entry at the entry of a function. And after the return instruction after the return instruction or after the function return, the stacks should be organized as follows.

(Refer Slide Time: 26:22)



EIP should point to the return address alright, ESP should point to the arguments that were pushed by the caller alright.

Student: Should not it be the last argument that going to or first argument?

All the arguments right; so, add function return the stack should be exactly as the caller left it before calling the function right. So, I am basically specifying a contract between the caller and the callee right. There is a caller which is calling the function and the callee which is been called right and there is a contract between the caller and the callee and that is what I am specifying here.

And esp should point to the arguments that are pushed by the caller. The function may have trashed the called function may have trashed the arguments alright. So, for example, if I call foo int a int b int star b, foo is free to change the value of a alright. And, when it returns the caller should not assumed that the value of a is exactly what he had sent it as it had sent it. It should assume that the stack is laid out in the same way, but it should not make assumption on the value of the arguments right, just a convention of the compiler alright. The other thing is EAX should contain the return value right.

Student: Excuse me sir why b pushed before a?

Why is b pushed before a just a matter of convention ok? There is no reason one reason potentially could be that there are some functions that actually allow multi variable number of arguments. For example, printf allows you a variable number of arguments and the number of arguments depends on the value of the first argument which is the format string right. So, to be able to support this kind of thing you know the first argument should be right close to the ESP and all the other arguments can be a variable.

Student: Sir anyways the caller has to remove all the argument and the return address from the stack right after there is return.

Sure.

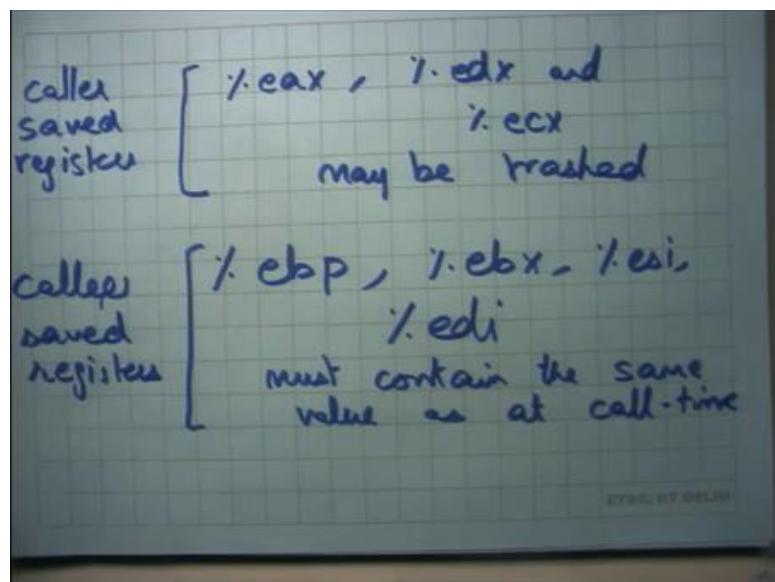
Student: So, while the callee or automatically pops all of them out?

So, question is why does not the callie just remove the arguments before returning, why does the caller need to do it alright.

Student: Sir.

See it is, it does not matter whether the caller does it or the callee does it; there is some the amount of work does not change right. The total amount of work does not change it is just easier to organize it in this way because the return address sets at the bottom. So, the callee just returns and then the caller can clean up the stack above it right. So, the clean the cleaning of the stack is done in the same order in which the stack was set up basically alright. EAX should contain the return value and then there is another convention which says you know certain registers can be overwritten.

(Refer Slide Time: 30:18)



So, for example, EAX, EDX and ECX maybe trashed by the callee and certain registers EBP, EBX, ESI and EDI must contain the same value as at call time right. So, here is another convention which basically say certain registers are allowed to be overwritten by the callee and certain registers must have exactly the same value as the callee as a as at call time right ok.

So, the terminology is these are called caller save registers and these are called callee save registers right. Basically, means the callee is free to trash these values registers. So, the caller must save them before making the call if it actually cares about those values right.

On the other hand, these registers the caller has the contract with the compiler that you know these are not going to be modified. So, it does not need to save them. If the callee does need these registers, then it is the responsibility of the callee to save them and then restore them before returning. So, if the caller and the callee follow this contract apart from that a function a called function is allowed to do anything else advance right. So, the called function can do anything it likes, but it should ensure that on return EIP will point to the return address that was pushed on the stack as it as the stack was setup on function entry.

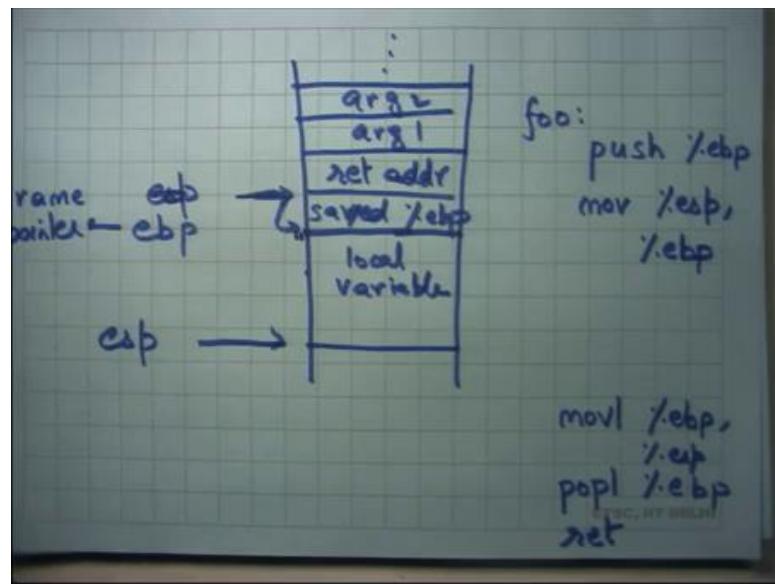
ESP will point to the arguments as I had set them up to be. It may have trashed the arguments I do not care EAX will contain the return value. These registers I may have overwritten, but these registers must contain the same value as I have as at call time fine.

If this contract is followed you know, I can call different functions and I do not and I can I can I can compile these functions separately and call them from one another and they will all work they can they can be forked together easily right. They need to follow this contract apart from that they can do anything else alright.

In your homework, we will ask you a question on why do you need separate caller and callee save registers, why could not, why does it makes to say sense to say that some part of the registers should be saved by the caller. And, some part of the registers should be saved by the callee, why cannot we just say all registers should be preserved right in which case all registers become callee save registers or conversely we could have said all registers should be can be overwritten maybe trashed in which case all registers become caller save registers.

Both these are options, but, but compilers usually take the middle path; half the registers are caller saved and half the registers are callee saved something to think about alright. We going to discuss that later.

(Refer Slide Time: 34:10)



Finally, on a function call so, at the entry of a function call the stack looks something like this, this is the return address this is argument 1, this is argument 2 and so on right and the function now gets to execute. Typically, what GCC does is it will also. So, the first instruction in the function would basically also save the current EBP value. So, called at the saved EBP alright.

And so, and it is so, it is going to push EBP. So, the first instruction of our function will typically be push EBP right. Let us say the first instruction of the function foo. The first instruction is going to push the current value of EBP, and it is going to set the current value of stack pointer to EBP or move ESP to EBP right. What is it doing? It is basically saying that alright. So, let us see what happens. I am going to push EBP. So, ESP becomes this and then I basically say move ESP to EBP.

So, what happens is EBP becomes this then the function may have some local variables and so, it is going to allocate some local variables. And, the way it allocates local variables are also in stack. So, one way to allocate local variables is keep pushing the values if they are initialized or just subtract a value from the stack. So, you know the size of the local variables; if there are like ten variables, if the ten integers in the local variables, then you will just subtract ESP by 40 right.

And so, ESP will point somewhere here, and you can de reference these local variables by using the appropriate offset in esp. And EBP is now pointing to the ESP value at the

start of the function right. So, in other words EBP is acting as a frame pointer right and you can use EBP to actually dereference the arguments. So, if you want to get to the argument number 10, then you just say EBP plus 4 plus 10 into 4 right that is argument number 10. If all arguments are integers let us say right. So, it acts as a frame pointer and the saved EBP is the frame pointer of the.

Student: Sir caller.

Of the caller right the saved, EBP is the frame pointer of the caller right. So, I made a call at the entry of the function the EBP value would still be the frame point of the caller. I first save the frame point of the caller, then I modify the frame pointer to my own frame pointer I modify EBP to my own frame pointer and now I start allocating local variables right.

So, that is at entry and before I return, what do I need to do? I need to do the opposite which is what let us say mov EBP to esp. So, I want to reset the stack exactly as so, I may have grown the stack a lot I may have allocated lots of variables I may have called lots of functions. And, now you know one single instruction way of resetting the stack is just to say move EBP to ESP because EBP is containing my frame pointer. So, if I just say mov EBP to ESP I have reset the stack back up and then I just say POPL EBP that is going to restore the saved EBP to my EBP. So, I get the callers EBP back into the EBP register and then I say return right. So, I go back to the caller right.

So, in this way I save the EBP value, I clobber the EBP value in my code; I write over it, I clobber it. And then I before I turn, I just pop it back. So, recall that EBP was a callee save register and so, the contract is that the EBP should be exactly the same as I gave it to you right. And so, this mechanism is actually doing that for you alright. So, this is a typical GCC behavior that you will stay save the frame point of the previous caller and you will initialize the new frame pointer and before returning you are going to restore the frame pointer of your caller.

What this allows you to do things like you know looking debugging things like you can look at the back trace of the of the current call chain. So, for example, if I want to so, say how many of you have used the backtrace command on GDB or something right.

Student: For the first homework.

For the first homework you have used it right. So, back trace tells you what the call chain is right. So, who called whom who called whom who called whom and how am I here and that is basically done using this frame pointer and saved EBP thing right? So, what it does is it looks at the current EBP. So, firstly, it looks at the current EBP and from that it knows where in which function I am, then it looks at the current EBP and from that it just displaces it by 4 to get its return address and from the return address it knows it is callee. It is caller right and from that it also knows the saved EBP of the caller and from that he recourses right.

So, he can just do that over and over. So, he gets he has the capability of getting the frame pointer of all its callers based on this recursion right. I just look at my EBP dereference, if I get the EBP of my caller I look at that EBP dereference I get the EBP of its caller and so on right. And plus, if I dereference plus 4 of it, I also get the return address. So, with each frame pointer, I also had the return address and that is that is how I basically have the call chain of the entire. So, that is how the back-trace command for example, is implemented on gdb right.

Student: So, sir how does this call chain ends?

How does this call chain end? Well you know, you could just have a terminating EBP value whether let us say it is a zero at some saved EBP is 0 for example, at main. So, you know you could just have some convention that this EBP is going to terminate and also notice that EBP was not strictly needed you know apart from debugging purposes, what am I using EBP for what I am using frame pointer for?

I am using it to reset the stack number 1, but resetting the stack I could have done anyways right because I know how many arguments I have pushed, I just need to pop that many arguments you know or how. If I incremented it by 40 or decremented by 40, I just need to increment it back by 40. So, I did not really need EBP to reset the stack, the second thing I am using EBP for is to get to the arguments right.

So, I say EBP plus 4 is argument number 1 EBP plus 8 is argument number 2 etcetera, but even that is not necessarily needed because the compiler knows that at this point esp is you know 70 bytes away from the frame pointer and so, because the compiler has this static information all it needs to do is add seventy to esp and then add you know 4 or

10 or whatever to get to the argument. So, EBP is not strictly needed, but it is a convenient way of doing things its edge debugging right.

Student: Sir.

Yes.

Student: Sir what if the called function somehow access the same EBP by also accessing local variables?

So, what if the called function the callee.

Student: Yes.

Overwrites the return address or overrides the saved EBP? So, that then he is violating the contract ok.

Student: But their called function might not know that that location it (Refer Time: 42:20) for it is accessing local variables, but somehow it is coded in such way that while accessing local variables it accesses saved EBP.

So, that is a bug in their function right. If you are supposed to be accessing your local variables and that and your program could potentially access the saved EBP that is a bug in your program in your function unless it was intended that way right. And in fact, such bugs have existed historically in a lot of our programs and such bugs have been used to.

So, the question is if I am accessing a local variable, let us say the local variable is an array and then I offset into an array and the offset happens to be bigger than the size of the allocation and so, now, I can actually clobber my EBP saved EBP or return address and things like that and based on that I can actually change the execution control flow. Well let us say I clobber the return address can actually jump somewhere else.

And this is a very common attack called the buffer over flow attack and such attacks have existed and basically this means that there was a bug in your code what you should have done was before dereferencing should have check the size right.

So, the code should really check the size before dereferencing always right that is the that is the safe guard that that all code must take and such bugs have existed, but people

have are much more aware today and such bugs are much rare alright. Finally, let us talk about the GCC let us come talk about a compiler workflow just to complete this discussion.

(Refer Slide Time: 44:09)



A compiler has a preprocessor which takes your source program which has this hash include or hash define directives and preprocesses them.

So, it is really a source to source transformation it takes C code which has this hash defined directives and just produces another C code. So, it actually does not look at the syntax at all it has does macro expansion it just replaces that hash define variable with its value or hashing it expands hash include with the contents of that file and so on right. So, that is a preprocessor. Then there is a compiler that takes a source file, parses it, and generates an assembly code right.

Assembly code is something that we have seen so far like move all. These are all these human readable assembly code like movl, addl and so on you know with their arguments. So, compiler basically parses C syntax and generates assembly code alright. Then there is an assembler that takes the assembly code which is the string which a human can read and makes binary code that a machine can read right that is an assembler.

The binary code is stored in dot o files like which are you know called object files compiler makes dot s files which are assembly files right. So, it gets assembled into binary code. These are called object files is machine readable code and then there is a linker which takes multiple object files combines them to make one executable. Let us say a dot out right. So, you could have multiple file multiple source files each source file gets compiled to an object file. These functions, these files could have independent functions like foo bar etcetera as long as they form follow the function calling contract it is ok. Foo can call bar in different files and the linker is going to ultimately link them all and make them and attach these all these references between foo and bar alright.

And then there is a loader takes a dot out and starts a turning alright. So, where have we seen the loader before? In the exit system call right. So, when you call exit system when you call the exit system call, you basically give the name of the executable file and inside the OS, there is a loader that will load that file.

The file should be in a certain format and that file is going to now get pasted into the memory space of that process and it will you will be started running by transferring control to the first instruction of that process right. So, that is the job of the loader so, that is how you know that is the workflow of a compiler and linker and a loader alright.

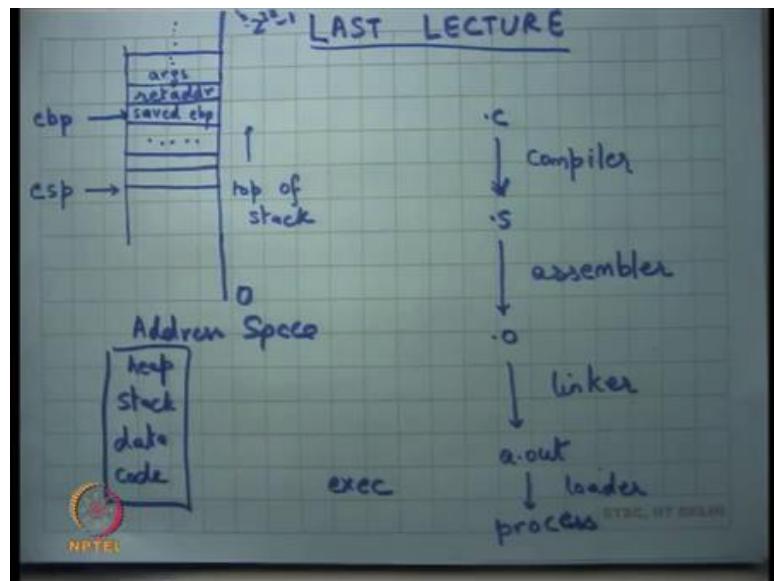
So, let us stop here and we will continue next time.

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

Lecture – 06
Physical Memory Map, I/O, Segmentation

So, welcome to Operating Systems, lecture 6.

(Refer Slide Time: 00:29)



In the last lecture, we were looking at how function calls are implemented and in particular we were looking at the stack, right. And we said look, so this rectangle that I have drawn here is really the address space of a process let us say and the stack is defined by this register called ESP, right.

So, this register ESP which means the stack pointer points to the top of the stack, right. This address space can be thought of as growing as you know going top to bottom which means let us say this is the lowest address 0 and this is let us say the highest address. And the highest address you know 32-bit machine will be.

Student: (Refer Time: 01:10).

You know $2^{32} - 1$, right. So, this is the address space and somewhere in the address space there is this pointer called ESP which points to the stack pointer. By

convention, the stack grows downwards. So, this stack the point where the stack pointer point is also called the top of the stack, right.

It is a stack in the sense that anything which is below the stack is junk. So, it can be overwritten. So, safely the program can start pushing data to the stack and does not have to worry about what is below the stack, right that is what a stack really means. But everything above the stack is useful and should be handled carefully.

It is not, it is not just your stack in the strictest sense because you can actually do random access to the middle of the stack, right as we saw last time. So, example if I just want to access this argument all I need to do is put an offset to ESP and I will be able to reach this argument, right.

Similarly, I can put an offset to EBP, and I can still reach this argument, right. So, you know it is a choice of the compiler. Typically, he will also maintain another pointer called the EBP which is also called the frame pointer and which points to which is equal to the value of the stack pointer at the entry of the function, right.

And at that point we as a convention compilers like gcc also save the previous EBP that is the saved EBP and so EBP points to the frame pointer and the first value there will be the saved EBP, the plus 4 offset value will be return address and so on, and then there will be the arguments. So, you can you know access these things using offsetting EBP.

And now all this can be my functions local bodies data for example, the local variables that I allocate on stack, all right. Above these arguments there will be more local variables of my caller, right.

And we also said look I mean this EBP pointing to saved EBP allows us to do back trace because we all I need to do is look at the EBP, from there I will get the saved EBP, so that will give me the frame pointer of my caller and so on and from each of these I can also get the return address, so I can get the whole call chain till for example, I reach the topmost function let us say main or something, all right, ok.

Then we said you know let us look at the workflow or the tool chain that allows us to build a program. So, we let us say we write our programs in high level language like c.

There is a compiler which compiler set into a dot s file dot. What is the dot s file? It is an assembly file which has all the instructions in assembly format or in assembly syntax.

Assembly syntax is just a human readable representation of machine instructions, right. So, this is a human readable representation of machine instructions. And assembler converts human readable representation to a machine-readable representation, so it converts it into actual binary format which a machine can read when it executes.

Then the linker can link multiple dot o files to make one a.out file. For example, you can call printf in your function, in your file, in your dot c file, but printf does not need necessarily need to be defined by you, right. So, printf could have been written by somebody else, but if you link all these things automatically things get patched so that the right printf gets called.

The a.out file then gets loaded. So, there is a loader which loads the a.out file. So, a.out file has to be in a certain format for it to get loaded and then it becomes a process, it becomes the running process, right. So, we said at the load is typically implemented in the operating system.

For example, when you make the exact system call, you know the operating systems loader comes into action it loads it looks at it reads the parts the a.out file, paste it into the processes address space, right. So, let us say this is the address space of the process, address space. It looks at the a.out file and the loader is going to paste the contents of the a.out file into the address space.

For example, that will paste the code into the address space it will post all the initialized values of the global variables into the address space, right and it will initialize a stack, right. So, it will allocate some space for the stack and add it will initialize the value of ESP, all right and so a process gets created.

And now initializes the value of EIP with the first instruction that needs to get executed which is also, this information is also encoded in the executable a.out and so now, the process starts running, right. So, from a.out it got the code and it got let us say the global, global data, the global variables of your program and all the local variables will now get allocated on stack let us say.

And then it also initializes the heap, right. So, you have heard of the stack and the heap before? In our courses, ok. So, let us, so the address space is let us say divided into you know code, data, stack, and heap, all right. I mean the layout need not be necessarily in this order, but roughly speaking this is the these are 4 different parts of the address space.

The code is basically the code that gets to run and as we said even code lives in memory or code lives in the address space of a process and code is obtained from the executable a dot out. Similarly, data refers to the global data of the executable and once again data is again obtained from initialized from a dot out, right.

The stack is initialized to some empty space, right and so some empty space is allocated in the address space and the stack pointer is initialized to point into the stack, so that now when the program runs it can actually start pushing and popping from the stack, right.

And then there is this thing called heap. Heap is basically all the other space that needs that the program may need to allocate. So, for example, if you want to allocate a data structure like let us say a linked list or a binary search tree or something then you would use you know neither stack nor global data are use or write the, right places to do it because stack grows only in a certain way data is fixed size you want variable sized dynamic on demand creation of memory that is what is called heap, right.

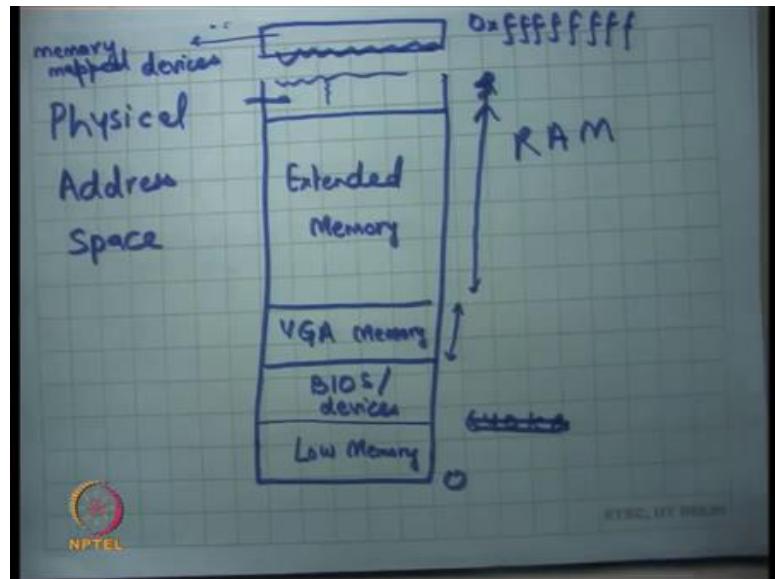
So, heap is basically an area in the address space from where you can with a program can request memory using calls like malloc or free memory using calls like free, right. So, that is what operates on the heap, ok.

So, the code and the data are initialized by a dot out. Stack and the heap are just initialized to you know empty address spaces with appropriate pointer pointing to the right things. And we are going to look at how malloc etcetera get implemented later on but let us just look at the stack for now and understand that, ok.

So, that is the address space of a process and you know I have not yet talked about how the address how a process, how the address space actually gets implemented. We have so far been assuming that every process has a private address space and we have talked about how the address space gets initialized by the loader and how it gets used etcetera and, but how does the address space get created that is what we want to talk about in the next couple of lectures.

But before we do that let us look at the physical address space that is available to the OS, all right. So, what is available to the OS?

(Refer Slide Time: 09:03)



So, let us draw the address space which is which I will call the physical address space. It starts at 0, all right and potentially goes all the way up to $2^{32} - 1$, all right. And let us look at you know, so let us look at a real world setting and let us see you know how does the physical address space look to the OS when the OS boots.

So, when the OS boots, he sees some memory and he can use addresses to access that memory and that you know that is what we are looking at. So, what are what address is referred to what parts of the memory. So, you know there is something called low memory right, till 640 KB then there are bios slash devices, right. Not necessarily in this order, but let us forget about exact numbers, but let us just say you know what all exists. So, there are bios and devices.

Then there is you know VGA memory. So, VGA memory is special memory. If the OS writes to that memory it actually appears as you know it actually goes to your display, right. So, let us say there is (Refer Time: 10:52), you know there is some area of memory which is basically called VGA memory and whatever the OS writes to this area of memory there actually goes to this place.

So, it is actually you know mapped into the display it is not real memory. And then there is what is called extended memory and the size of this is basically dictated by the size of your RAM, right. So, when you say I bought, I buy a you know I bought a 1 GB machine, so the size of the extended memory is basically 1 GB, right.

And then at that, so let us say the extended memory is to this point then everything above it the address space is unused which means that if the OS says I want this address it is going to get an error, right. The hardware is going to generate some kind of an error signal.

It is going to say error interrupt and that is what you commonly see as a bus error, you know if you have ever seen the bus error in your programs; the bus error basically means that an address was generated which does not have a mapping in the physical address space, right, ok.

And then you know somewhere on the top there what is called memory map devices. What these are basically, these are addresses that are backed not by physical RAM, but by devices, you know even think of some device and the idea is that when you write to this device to this memory address it actually goes to a device, right, it goes as a let us say command to the device and you read from that memory address it actually becomes a read from the device, right.

So, the device will for example, have certain registers. So, let us take for example, the printer, right. So, let us say the printer has certain registers which say you know start a command and stop command etcetera and these registers are actually mapped in the physical address space.

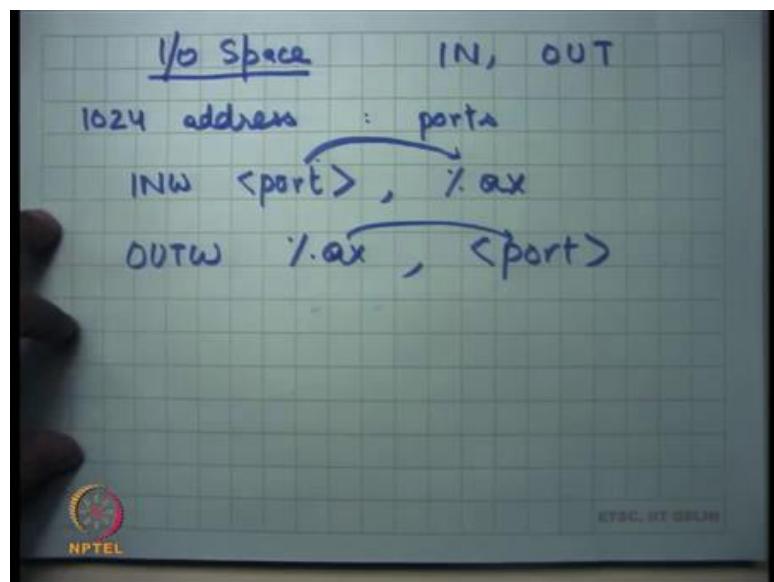
So, you know I am just taking examples, but the exact mappings will be determined by some kind of a manual which the printer manufacturer gives you, but basically the way it will work is that the registers are mapped in the physical address space and the OS is can read or write to this a physical address space to send or receive commands and data from to and from the device, right, ok.

So, this is part of the physical address space, but it does not necessarily behave like memory, right. What are the semantics of memory? That if I write something to address

x and then I later read something from address x with no intermediate write to the same address then I am going to get the same value.

But this address space will not necessarily follow those semantics, right. So, in other words writes and reads to these addresses may have side effects, ok. So, they do not necessarily behave like memory, but they share the physical address space, ok. So, that is one way of accessing devices.

(Refer Slide Time: 14:07)



Another way of accessing devices are what is called using IO space. So, x86 also has special addresses which is, special a separate space for IO which is called the IO space, which is accessed using special instructions like IN and OUT. So, last time I said that there are special instructions for IO called IN and OUT. So, let us see how they work in action, right.

So, the way it works is the IO space is actually made of 1024 addresses. So, it is a pretty small space actually compared to the 32-bit space that we had for memory, right. So, it is only 10-bit space for the IO. And so, and these addresses are also called ports, IO ports, right.

And I can say I can use commands like in from a port, right to let us say register, so let us say the register is ax. So, let us say it is I want to read a 16-bit value from this port to register ax this is the instruction to do that, ok. So, once again as we saw in the memory

mapped case there were certain registers of the devices that are mapped to certain addresses of the physical memory.

Here certain registers of the devices are mapped to certain addresses or the IO space, right. The difference being that in that case I could just do read and write to that location or in fact, I could execute any instruction on that memory address like increment, decrement, subtract, add, move etcetera. In this space I can only use these special instructions called IN and OUT, right.

So, IN basically says read a value, read a byte or read in this case read let us say a suffix with W, in this case it says read two bytes from this port to this register ax, right or I could say OUTW percentage ax some port which basically says write these two bytes from ax to this port, ok. So, this is there is a way to access devices.

And let us look at a real example of how something like this works. So, we have seen two ways of accessing devices one is using the physical address space of memory and the second is using the IO space which we call also called ports, all right. So, let us look at let us say the printer.

(Refer Slide Time: 16:37)

```
#define STATUS_PORT 0x379
void lpt-putc (int c) {
    POLLING ← while (((inb (STATUS-PORT) &
        0x40) == 0);
    TROBE ↑
    outb (DATA-PORT, c);
    outb (CONTROL-PORT, STROBE);
    outb (CONTROL-PORT, 0);
}
INTERRUPT
```

So, you know I will give you some example code or some pseudo code of a printer driver which basically wants to put a character putc to a printer lpt, right line printer let us say. So, I want to I character to the line printer and let us say I want to implement a function

in C which puts this character c to the printer, all right. And let us see how something like this could be implemented.

So, I could say while (inb (STATUS_PORT)), all right. What am I doing here? inb is another function which internally is just executing this instruction called inb, right on this port called status port. So, let us say you know status port has been defined here some value let us say, right. It is some number less than 1024 and the printer manuals say that this is the port on which I will basically be accessible to you, right.

And so, now the software can say make can execute the instruction in b on this particular port and get the data on that byte in a particular register and then you know it can execute its regular code to check the value. So, in this case I am actually you know I am writing C code, but you can you know imagine converting this to assembly, where you just read you execute the in instruction on that port, you get the; you get the byte and then you compare the byte with this particular value busy, right.

Let us say busy is also some number and bit and you come and you AND it, and till its busy you keep spinning, right that is what it is saying. So, it is saying till the status says I am busy keep spinning. So, it keeps spinning till the status is busy, right.

As soon as the status becomes not busy it is going to break out of the loop, all right, ok. Then at this point I know that the printer is not busy, and I can let us say send character c to the DATA_PORT, right. I am writing these as functions, but these functions are basically just single instruction functions that just execute the outb instruction, ok, all right.

And then I put a data value on the data port and then I may need to tell the printer that look now sample the data, right. So, I put some value on the data port and now I need to let us say tell the printer that you know please pick up this data I put the data in your register, now pick up the data and let us say the printer was edge triggered which means you know there is some strobe signal in the printer which picks up the data on every edge or every transition of the strobe signal.

So, let us say I do that by saying by putting, so there is a CONTROL_PORT let us say. So, let us say the value of STROBE was 1. So, I put 1 to the control port and then I put 0

to the control port. What this does is it simulates an edge on the control port and that is when the printer is supposed to pick up the data from the data port, right.

So, an example of how let us say a driver could be implemented, right. So, first you check the status port waited it for it to become free, then you wrote some data to the data port register and then you strobe the control port for the printer to pick up that data.

And now the printer may say I am going to be busy for the next few milliseconds or whatever, let us say its printing that data, in which case if there is an extra character that needs to get printed it is going to be now the CPU is supposed to wait till the printer is actually ready to take data, take the next byte of data, ok.

Now, clearly let us say there is just one printer and you know, so the operating system should be the only one that is that is actually executing code like this, right. It cannot be like a process cannot be executing code like this, right, because you can imagine if multiple processes try to do the same thing then there will be chaos, right.

Because both of them are going to see the printer is free and both of them are, I am going to try to write to it and so the printer state machine is going to get confused. So, there is going to be there has to be, there has to be some organization such that the printer state machine gets followed properly and one way to do that is to use the OS as an intermediary.

So, the process is not going to access the printer directly, process is going to use system calls like read and write and the OS is going to convert the system calls into code like this, right. Also notice that this kind of code involves this while loop. While the CPU is executing this while loop the CPU is essentially busy, right. I am basically just checking the status port to see if the status if the printers got free.

A more efficient use of my resource could have been that I use the cp; if you know the status port is its busy I figure that out and I start executing something else and there is some way for the printer to let me know that the status port is now free and, so that is when I start executing the rest of the logic, right. That would have been the more efficient way of doing things, right, in some situations.

So, this is called polling. Polling means I keep checking, perhaps at periodic intervals whether the condition I am looking for has been, is because has become true or not, right. As opposed to the other approach which I just described which is called interrupts ways approach in which case I tell the printer look interrupt me when you become free, right, and so and now I start going doing something else. And now when the interrupt becomes when the printer actually becomes free it interrupts me and I check again whether it is actually free and then I execute the same logic.

Student: Sir, in this called busy variable like, busy is the variable or a constant?

Busy is a constant.

Student: Sir, so if we have just (Refer Time: 24:31) and (Refer Time: 24:32) so, like why are we doing bitwise AND (Refer Time: 24:37).

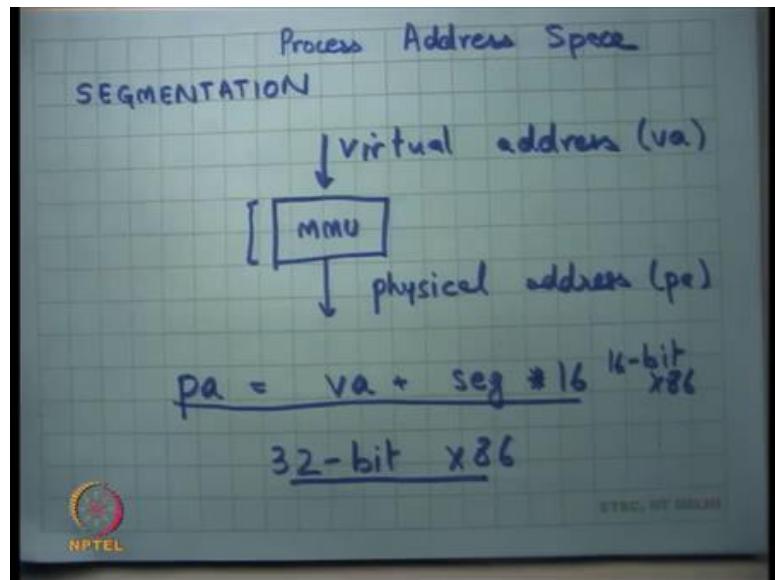
We are just checking if a certain bit in this value status port, in the value returned by status port has been set or not, right. You know for simplicity you could even ignore this and say you know let us say the specification says that the status port is going to be either fully 0 or fully 1 depending on whether it is busy or not, right. I was actually just; so, busy is just a way of saying that I am checking 1 bit as opposed to the entire byte, ok, all right.

So, polling versus interrupt, these are always two options in system design, right. So, one entity wants to access another entity, whether I should keep checking whether that entity is ready to receive my message or whether I should use an interrupt based mechanism and it is not necessarily that one is better than the other. Interrupt comes with its own cost, right.

Because if I set up an interrupt on the printer, the printer is going to in future invoke an interrupt handler and there will be some cost associated with executing that interrupt handler. For example, if I expected that the status port will become free very soon, you know let us say in the next 20 to 100 iterations I am very likely to find in the status port to become free and actually cheaper to do polling, all right. On the other hand, if I know that the status port is going to be free after a long time let us say 100,000 cycles or million cycles then it makes sense to do interrupts, right.

So, it is really am choice that an OS has to make depending on the device characteristics and the workload characteristics whether to use one or the other. And we are going to see more examples of this design choice later when we talk about real examples, ok all right. So, now, let us talk about process address spaces.

(Refer Slide Time: 26:45)



So, far we have looked at physical address space and IO address space, right. Now, let us look at how an OS implements process address space. So, once again what are our requirements for a process address space? Firstly, it should be private which means nobody else should be ever able to access it, right. Second, it should be protected which means a process should never be able to step out of its address space.

So, it should not be able to access anything outside the address space, right. So, let us just work with those two sorts of requirements. And, one way to implement address spaces is what is called segmentation, all right. So, how does; what a segmentation? So, we saw when we were discussing the x86 architecture 16-bit, we said there are these special registers called segment registers CS, DS, SS, ES, right.

So, they were basically being used to add. So, there is an MMU. Here is an address that the program gives you and here is the address that actually goes out on the virus to the memory, right. So, let us get say that the address that the program gives you is a virtual address and the probe and the value that actually goes out on the wire physically is called the physical address.

In the 8086 machine that we saw this MMU was doing a simple addition. So, you know let us say this is va and this is pa then on the 8086 pa was just equal to va + segment * 16, right and that is really not enough for you to be able to provide any kind of protection or anything of that sort.

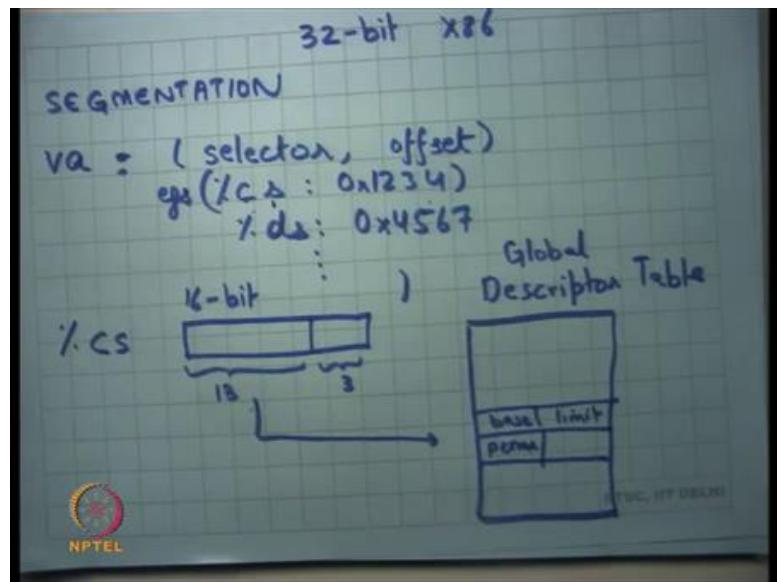
It was just a matter; it was just a matter of convenience that you know you can access more memory with less number of virtual addresses. But now we want the MMU to be smarter in the sense that the virtual address of a process should get translated to a physical address such that it only accesses a memory that it is allowed to access, it never steps over other person's memory and so on, right.

So, let us look at how such an MMU is implemented. And now and what I am going to look at is segmentation on 32-bit x86. So, this was 16-bit x86. This is 8086 which we saw. And now let us look at how processes are implemented or how segmentation works on 32-bit x86, right.

When 16-bits machine processors like 8086 were designed at that time multi-processing or the ability to have multiple processes which do not trust each other was not that important because you know workloads had not evolved to that stage at that time.

So, that time they just use the simple segmentation hardware to be able to do MMU, but by the time 32-bit and very soon you know multi-processing was needed and so they needed more kind more hardware in the MMU to be able to do that, all right, ok.

(Refer Slide Time: 30:27)



So, let us look at 32-bit x86 segmentation. So, once again a virtual address is a pair of segment selector and an offset, right for example, CS: 1234, right or DS 4567 and so on, right. So, that is what of, that is how a virtual address is specified. It is specified using what is called a segment selector or the name of a segment register like cs, ds etcetera and an offset which is the 32-bit offset now, in the 32-bit world.

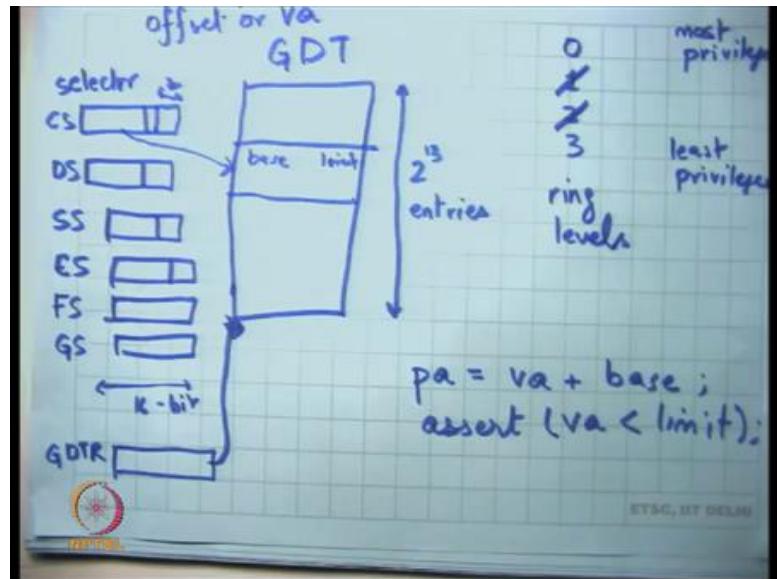
And once again in the 16-bit world it was very simple I just used to multiply this by 14 and add to it, but now we were going to do something, something more complicated because we want to provide protection, all right. So, the selector let's say CS is a 16-bit register of which the top 13 bits are used to index into a table which we call the descriptor table, let us say it is a global descriptor table into an entry which contains things like base, limit, permissions and so on, all right.

So, what is happening? If the application specifies CS: 0x1234, I am going to look at the CS register I am going to take the top 13 bits of the CS register use it to index this table which we call the global descriptor table to get an entry. This entry will have values which say base, limit, permissions among others, right.

And now what the hardware is going to do is going to add the offset to the base before comparing it with limit. So, first it will compare the offset will limit to see whether you know this offset is allowed within this segment or not and then add it with the base and

that is what the physical address is going to become. So, the physical address will be offset coming from here plus base assuming that it was less than limit, right, ok.

(Refer Slide Time: 33:55)



So, let us see here is a global descriptor table let us say I call it GDT. Here our segment registers CS, DS, SS, ES and actually 32-bit has 6 segment registers FS and GS, these are all 16-bit wide, all right. The GDT itself has 2^{13} entries, ok. The top 13 entries, the top 13 bits of each of these points into the GDT and tells you which entry to use to do the translation, right. So, let us say there is an offset that is coming from the user and there is a segment selector, so this is a selector.

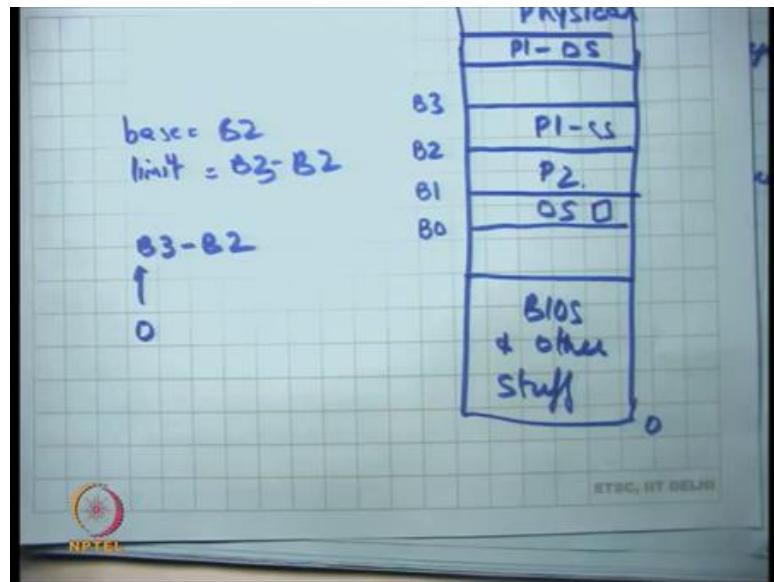
The selector the top 13 bits tells you which entry to look here look at here and from here I am going to get numbers like base, limit, and permissions. But let us just say base and limit for now. What I am going to do is I am going to say $pa = va + base$, where va is this offset. But also assert that $va < limit$, all right. So, it is giving you more than what the 16-bit does and the thing that is giving more is that it is also asserting, it is also checking that it is not overflowing, it is permitted limit, ok.

Student: Sir, it asserts some va or a pa?

It asserts some va, ok. Just a matter of convention, all right, ok. So, what am I going to do? You know if I wanted to implement address spaces, one way I could implement address spaces is I would set up the segment registers in such a way that they point to

memory regions which are allocated for that process and I will set up limits in such a way such that the process is never able to access anything beyond what I have allocated for him, right.

(Refer Slide Time: 36:57)



So, for example, what I could do is let us say this is the physical address space and now let us say this is 0, you know this is BIOS, and other stuff, but here is where the actual memory starts which is usable memory, right. So, from here I said ok, this region I want to give to process P1, right and this region I want to give to process P2 and this region I want to keep for myself, right.

So, what I am going to do is I am going to look at let us say this was base B0, this was base, this was address B1, this was address B2 and this was address B3. When I run when I scheduled process P2, I am going to set up the segment registers in such a way that let us say I want to run process P1, so segment register will P set up in such a way that base will be equal to B2 and limit will be equal to B3 - B2.

And so, when I run this process P1 when it uses a segment selector it can only access this area of the memory, right. So, the process itself will use addresses like 0 1 2 3 or you know any other address.

The legal addresses for a process in this case would be 0 to B3 - B2, right. It can use any of these addresses. 0 gets translated to B2, B3 - B2 gets translated to B3, anything in the

middle just gets translated to $B2 + pa$, right. So, the process will just use 0 till some limit and the OS is going, the hardware is going to converted to the appropriate physical address.

Assuming certain things a process will never be able to access anything outside this region, right. So, that is how I am going to implement address space. Notice that in doing so, a process does not have to worry about where it was placed in physical memory, right. The executable can assume that it is starting from address 0 or the address space is starting from the address 0 and it has some limit to it, right.

So, when so in the compiler, linker, loader its or the compiler, you know assembler and linker do not have to be worried about exactly where the process will be placed at runtime, right. The canonical there is there is some convention that the operating system is following, in this case the operating system is following that the address starts at 0 and goes up to some maximum limit and so the program safely can be put at address 0 let us say, right, irrespective of where it gets put in the physical memory later on.

Student: Sir, how can we know process needs this much address before the processes started from (Refer Time: 39:56) address space?

So, how do we know at load time how much memory to allocate for that process? In some sense we are limiting the space that the process may have, right that is a great question. Firstly, we know that when you load the, when the OS loads an executable it knows the sense of how much how many bytes the code takes and how many might data takes. It also has some default values of how much space to allocate for the stack, right.

And then it will have no, let us say in this scenario I could also say here is my default value of, here is the maximum value that you can allocate for the heap, right and so the operating system is forced to make this pre-allocation upfront at load time.

The process actually may be a very short process and may not allocate any memory at all in which case I just over allocated or the process may actually not need a lot more memory than what I allocated in which case I under allocated. And in both cases I will have to tell the process using some kind of signal like segmentation fault or something that you have trying.

So, in the first case there is no problem, in the second case you will basically tell the process that look this you are doing something illegal not allowed, all right. But of course, you know I am giving you a first flavor of how address spaces can be implemented, there are more advanced ways in which address spaces are implemented and that allow better more dynamic allocation of space to processes, all right. So, let us just; let us just look at this for the moment, all right, ok, all right.

So, plus actually here I have drawn different processes as different address spaces. By the way though if I wanted to some space to be allocated only for the OS, the OS could say you know this space will never be marked as base and limit for any process, but will be used as base and limit for its own functioning etcetera, right.

Here I am saying that each process must; so, one limitation of doing this base and limit way of doing physical MMU is that a process needs to be contiguous, right. One could relax that requirement by saying that look there are 6 segments. So, I should ideally allow up to 6 different contiguous segments, right. So, an OS could be more complex and say I am going to have you know P1, CS here and P1 DS here or something, right.

So, that way a process is allowed up to 6 different contiguous regions and so in that way it can have more dynamic allocation depending on what it does. So, example something like the code segment needs to be allocated statically in most cases. So, it can just get allocated statically and other things like stack can be allocated on demand or heap can be allocated on demand, ok.

But more interestingly let us understand how an OS can actually set up these segment selectors, but not allow a process to be able to manipulate these segment selectors. So, the hardware has to make some give some hardware support to the process. So, that the process is not able to overwrite the segment selector, right or the process is not able to override the global descriptor table and manipulate the base and limit, right, ok.

So, firstly, how is the GDT computed? So, there is another register on the hardware on the chip, which is called the GDTR global descriptor table register, which points to the base of the GDT in memory. So, this GDT itself is saved in memory, right and so GDTR points to the GDT in memory. And what the hardware is going to do is going to take the selector take the 13 bits index, the pointer at GDTR add these 13 bits to GDTR and

multiply them by whatever the size of an entry is and then use that to get base and limit, ok.

Student: Sir, what have been we have more than 2 to the power 13 processes like because totally child process also. So, what (Refer Time: 44:34) more than 2 to the power 13 processes?

What if there are more than 2 to the power 13 processes? It is ok, right. So, we are I am going to discuss how exactly it is implemented. Just.

Student: Sir, what about the remaining 3 bits (Refer Time: 44:47).

What about the remaining 3 bits? Let us just hold on and we are going to discuss how they are used, all right.

(Refer Slide Time: 44:59)



So, firstly, the GDTR itself should not be manipulatable by the process. So, a process should not be able to set a value of GDTR because of the process is able to set the value of GDTR then he can pretty much do anything he wants, right.

So, to be able to do that firstly, the processor must support two modes, privileged and unprivileged, right. Privilege mode is the god mode, can do anything, right. If you are running in the privileged mode, you can do pretty much anything. Unprivileged mode is

the peasant mode where you know the god tells you exactly what you are allowed to do, right.

Now, the process runs in the peasant mode and the operating system itself runs in the god mode, right. The hardware has certain restrictions on what instructions can be executed in god mode and cannot be executed in, there are, there are certain instructions that can be executed in god mode but cannot be executed in peasant mode.

And the instruction that sets the GDTR, load GDT is an example of one such instruction, all right. So, the lgdt instruction which actually loads the GDTR can only be run in god mode not in the present mode, right.

So, you are going to as you can imagine that you are going to run the process in the peasant mode and because the if the process is running in the peasant mode, the process even if it executes the lgdt instruction what is going to happen? It is going to create an exception, all right. Just like a segmentation fault is going to create an exception, the operating system is going to get notified and it can for example kill the process, saying that you are trying to do something illegal.

On the other hand, the operating system is free to execute this instruction that is number one, all right. So, the GDT can only be executed set by the OS and not by the process, ok.

Second, the segment registers should they be allowed to be overwritten by the process?

Student: No.

Well, one answer is no. But, actually on x86 a process is allowed to overwrite its own segment registers, right. For example, it may be using you know the compiler may be using things like it I want to shift my code segment somewhere, right. So, the architecture designers did not want that user should not be able to set up its own segment registers because segmentation, the segment registers are used for multiple purposes apart from this protection.

So, user is allowed to override the segment registers, but then what protects it? What disallows it from doing something wrong? It can only set up these registers to one of the allowed values in the GDT, all right.

So, what it can; so, what an OS can do is that it can engineer the GDT in such a way that only certain values, certain base and limit values are present in the GDT and so even if the program changes its selectors it can only change it to one of the allowed values, right. So, if there is no base and limit for the OS address space in the GDT then it will never be able to change to that. It will never be able to access the OS's address space, all right.

Student: Sir, we can still change the segment to another process (Refer Time: 48:42)?

We can still change the segment to than other processes base and limit. No, even that is easy, right. Before a context switch to this process, I am going to switch the GDT such that it only contains entries of that particular process, right.

So, at the context switch time before I start this process running, I will remove the GDT of the previous process or I will modify the GDT such that the previous processes entries are not there and the new processes entries are there and only the new processes entries are there. So, the new process is allowed to freely change his segment registers between these new entries, but he is not able to access any other processors memory or its own memory or the OS's own memory, all right.

Convincing that this is a this is a way of ensuring that applications have private address spaces and do not are able to write on another application that is space and are not able to write on the OS's address space.

Some more facts. How does the processor know what is the current privilege level? The way an x86 processor decides what is the current privilege level is by the last two bits of the CS register. So, the last two bits of the CS register tell him, tell the processor in whether I am working as god or whether I am working as peasant, all right. So, the convention is that 0, so they are two bits. So, the you have 4 possible values 0 1 2 3, all right and 3 is the least privileged and 0 is the most privileged.

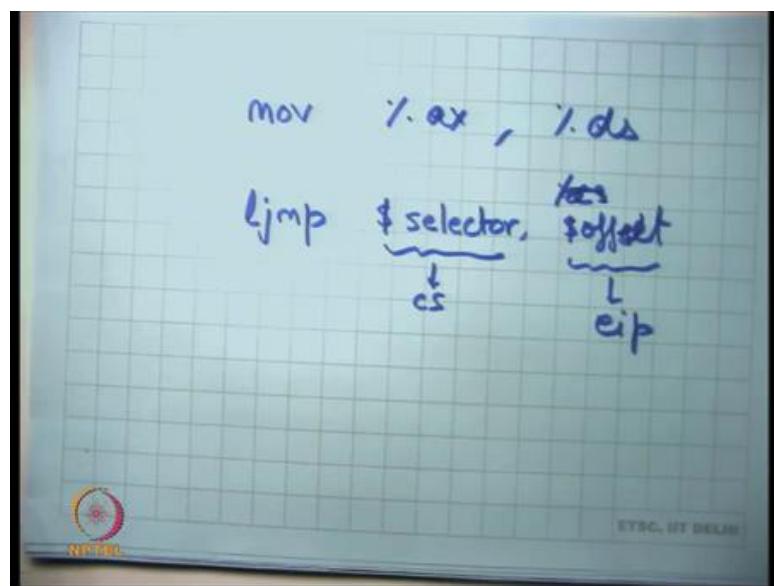
Most operating systems just use 0 and 3, do not use 1 and 2 at all, right. So, the operating system set, when the operating system is running the last two bits of CS are set to 0 and when the application is running the last two bits of CS are set to 3, right. So, 0 and 3 is basically, decides whether I am running in this mode or that mode, all right.

Student: Sir, when the process can modify CSO, if it modifies these bits than it will if the?

Interesting. So, a great question. But a process can modify its CS. So, if it modifies its CS then it can you know change its privilege level at will. So, the hardware also makes sure that only certain types of modifications are allowed, ok. So, when you modify the CS if you cannot go from a less privileged mode to a more privileged mode we can go in the other direction for example, right, ok.

But in some cases, for which we need for implementing system calls which you are going to discuss later but let us just assume for now that a process cannot in general just upgrade its privilege level by modifying CS, right. If it tries to do that, so you can change the CS and the way to change the CS. By the way how can you change that how can you change DS? So, there are instructions how you can change segment registers?

(Refer Slide Time: 52:09)



The instructions like move, a register like ax to the segment register. So, this is a valid instruction. A process is free to execute this instruction to change a segment register. But CS is an exception you cannot just execute this instruction on CS, just like you cannot change EIP directly you cannot change CS directly.

So, the way to change CS is basically what is called a ljmp instruction, all right. And ljmp basically say takes arguments selector and offset, right. So, we have seen the jump

instruction before, where we just say jump to an address it just changes EIP to that address. ljump takes two arguments select and offset and the semantics are that it will set the selector in to CS and set the offset into EIP, right that the semantics of ljump.

And when it executes the l jump instruction only certain values of offset are permissible. So, for example, if I am executing in at privileged level 3 and I execute l jump to a selector which has privilege level 0 in it, that is not allowed, it will cause in the hardware to generate an exception and the OS will come into action, right. So, only certain values are allowed. So, basically you are not allowed to go from a less privileged mode to a more privilege mode, ok.

Finally, what prevents an application from just modifying the GDT? So, GDT lives on memory, right. Why cannot an application just say fine figure out the address of GDT and just write something there?

Student: Privileged instruction.

So, one answer it is you require a privileged instruction to do that. Is that right? Only need a memory move instruction, right just some move instruction is going to suffice to be able to write to a memory address.

Student: Because only base is privileged more (Refer Time: 54:14).

Right. So, the GDT is stored in the region of memory which is only accessible to the OS and not to anybody else. So, when I have this figure and I say this is the physical memory. So, GDT is probably going to live somewhere here, right. And because this area is not mapped into the address space of any other process no process can actually modify GDT, right.

So, we are looked at protection. We saw that a process has different privilege levels, let us say there are two privilege levels at least, unprivileged and privilege. Certain instructions can only be executed in privileged mode the loading of GDT is one such privilege instruction.

Then we said that the segment registers are, can are free to be changed by the application, but they can only be changed to certain values and those values are dictated

by the values in the GDT, all right. And you cannot just lower your; lower your privilege level or upgrade your privilege level or lower your you know, so low your ring level.

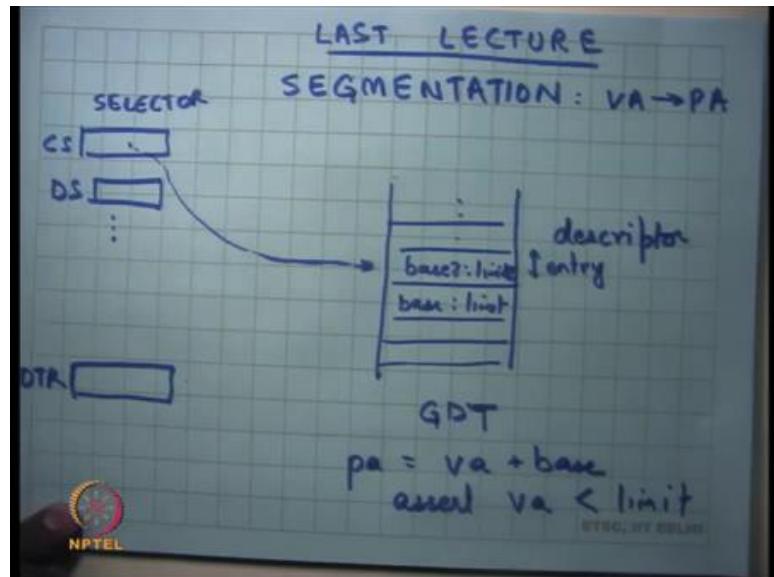
So, this 0 1 2 3 are also called ring levels, right. Lowering a ring level means upgrading your privilege level. So, an application cannot just lower its ring level and so that is another thing. And third, finally, because GDT is a protected structure, a protected structure must live in a section of physical memory that is not mapped in the address space of any other processes memory and only in that is space or the OS memory, ok.

Let us stop here. And, we are going to continue this discussion next time.

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

Lecture - 07
Segmentation, Trap Handling

(Refer Slide Time: 00:23)



In the last lecture, we were discussing Segmentation. Segmentation is a way to implement process private address spaces, and to divide physical memory into virtual address spaces for the processes right. So, we said there are you know in some way segmentation provides mapping from virtual address to a physical address right.

A virtual address is what the process is, and the physical address is what actually goes on to the via to the memory right. And so, you know we said that the there is some logic on the hardware of the processor, which we call the Memory Management Unit or MMU which does this translation. And the particular mechanism that we are looking at right now is segmentation all right.

So, this is how segmentation works. There are multiple segment registers. Every memory address or every virtual address that an application specifies which could be a specified using direct, indirect, or displaced modes which we have already seen in the instruction encodings. Whatever address there is it has to be prefixed with a segment id, so that

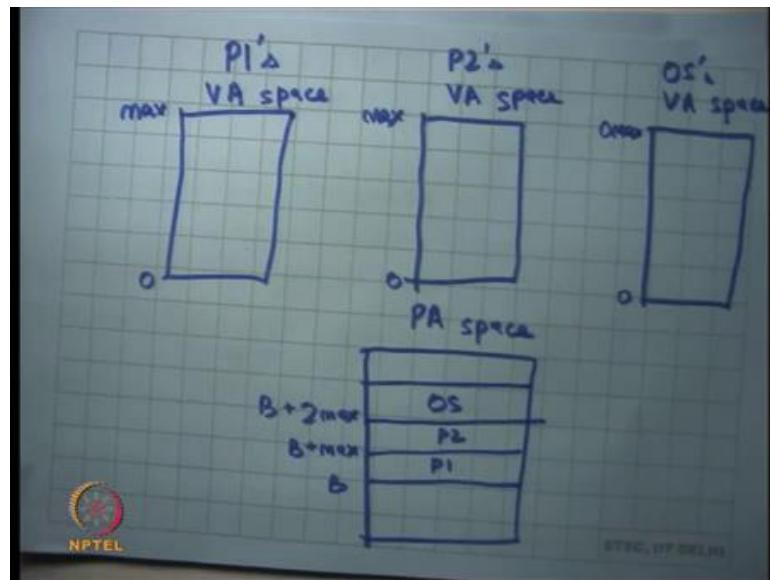
address that is computed by an instruction is actually the offset inside the segment that is specified by the segment id, and the segment id would be CS, DS, CS whatever right.

The segment gives you a pointer into a structure called the global descriptor table which also lives in memory, right. The address of the global descriptor table is stored in, yet another register called the global descriptor table register right. So, what the hardware does on each instruction is that it will dereference GDTR to get the address of GDT.

It will add the value of the segment selector, so these are called segment selectors, it will add the value of the segment selector to the base of the GDTR or to GDT into whatever the size of the entry is. So, this is one entry of the GDT right to get another appropriate segment descriptor.

So, each entry inside the global descriptor table is called a descriptor, a segment descriptor. What the segment descriptor holds are values like base and limit right. And what the hardware is going to do is it is going to compute pa as va + base, and it is also going to check that va should have been less than limit ok. So, the hardware is doing this at runtime on each and every instruction all right.

(Refer Slide Time: 03:26)



So, if you just look at it, if you we just look at what it is actually implementing, if this is the address space of process P1, so let us call it P1's virtual address space P1's VA space and this is P2's VA space right, and let us say this is the OS's VA space all right.

And let us say this is the physical memory all right, so let me call it the physical address space PA space, and we looked we saw last time how the PA space is organized there as bios, there are devices and then there is actual physical memory inside the PA space right. And now what the OS does is basically sets up the segment descriptors in such a way that you know, each of these maps to different regions of physical memory.

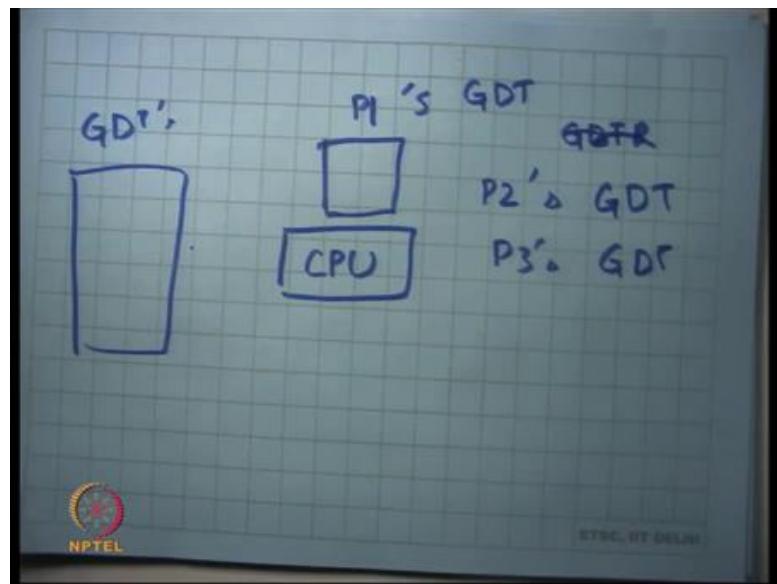
So, let us say you know P1's VA space is here, so you know let us say P1's VA space starts at 0 and goes up to some maximum value max. And let us say all the processes just for simplicity I have a maximum value of max that is maximum at address space a process can have, and let us say you know the OS also has 0 to some you know 0 max, OS is max right.

So, then you know I could not the OS can decide to place P1 anywhere he likes, so example he starts at B and so and, so what does he set his basin limit to he sets a base to B and limit to max right. Similarly, P2 can live here, when this case becomes you know B + 2 max let us say and he can put the OS somewhere, so let us say OS put somewhere all right.

So, this is how the physical address space looks like when I am using segmentation, and this is what the physical address the virtual address space of each process looks like. Notice that the virtual address space is completely independent, or the naming of the virtual address space is completely independent of where it lives in physical memory right.

So, it gives you independence, so it gives independence between the linker and the loader. The linker at link time does not need to worry about at what address I am going to get loaded right, the OS can make that decision at runtime all right ok. So, one way for an OS to implement virtual memory is that it can say so let us say here is a CPU all right.

(Refer Slide Time: 06:16)



So, an OS could say at this point process P1 is scheduled on the CPU. So, I am going to switch my GDTR, so you know let us look at some ways in which an OS could actually allow multiple processes to live to coexist in the system. So, one way is you know let us say I have one GDTR per process possible right. A GDT is 2^{13} entries, each entry is let us say 8 bytes, so roughly 2^{15} bytes that is 32 kilobytes roughly ok.

So, you have 32 kilobytes of space per process just for doing implementing virtual memory all right. So, I could have you know another GD, so I could have P1 GDT, I could have P2 GDT, P3 GDT and so on right, so that is one way of doing it. And each time I decide which process is going to get to run on the CPU, I just switch the GDT all right, so I switch the GDTR to that processes GDT make sense, no question.

Student: Sir, why do we need different GDT for each process?

All right, so why do we need different GDT for each process, it seems wasteful. Another way to do it is let us have just one GDT, and each time I switch a process I just change the entries in the GDT now that is another way of doing it, so these are all ways. The second way is obviously superior to the first way all right, because I am not wasting space, I just have one GDT and I can just manipulate the entries on each context switch.

Student: (Refer Time: 08:30) very large.

Very large yes. I was just trying to you know give you a perspective on what all can be done right. So, basically if I want to schedule this process, I load his address space; if I want to schedule that process, I load that process address space and so on ok, all right. So.

Student: (Refer Time: 08:54).

Yeah.

Student: You say that how every process changes its address space and can only change the segment registers to specific values that is how it will be controlled. So, the OS knows for which process, which segmented, which values will be allowed, they look they are not a person it has to organize all those values in that way.

All right. So, let us just review protection how does protection work? The GDT itself lives in the address space of the OS. So, an application is not allowed to manipulate the entries of the OS or and so it is not allowed to manipulate the entries of the GDT all right. And secondly, I am not able to load another GDT a process is not allowed to load another GDT, because the instruction to load the GDT can only be executed at privileged level.

And so the process the processor has the concept of unprivileged and privilege modes, and so the process if its running an unprivileged mode cannot execute this instruction, so I cannot change the GDT; I cannot reload the GDT, I cannot change the GDT, because GDT is not mapped in my space and that is enough right.

Now, what the OS needs to do is it needs to do some kind of bookkeeping that which area, how much memory is allocated for which process and where I have allocated it and so that kind of bookkeeping the OS will have to do all right. So, for example, the OS will keep it keep information like what is the base and limit of all the processes that live in my system all right.

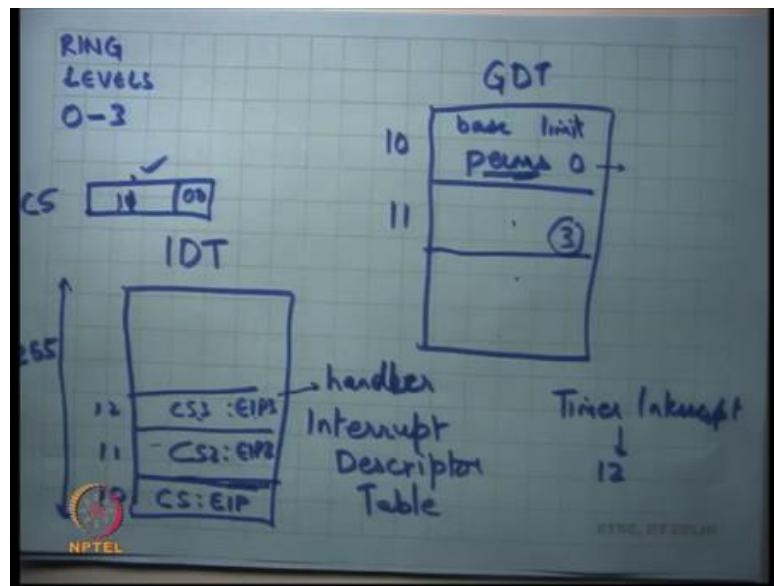
And so, when I switch to that process, I am going to pick up that base and limit and put it in my GDT all right, and I am going to start that process right. Where are where these data structures being stored, which contain the base and limit of each process.

Student: In the OS address.

In the OS address space right. A process should not be able to manipulate these things either right, so these things also need to be live living in a protected address space of the OS all right, The other thing is that sometimes so now, I am going to talk about how things like system calls are implemented. So far we have talked about a process having its own address space and a process being confined to his address space, so he cannot do anything else right, but we have also said that you know a process cannot live in isolation, it also needs to talk to the OS sometimes for example to make system calls.

And so, and also the OS needs to take control from the CPU at periodic intervals to be able to do scheduling and preemption. So, for that even though there are times when you would want that both the OS and the application can should be mapped in the address space. So, both if the so when the transition happens from application to OS, the OS is address space should also be mapped, so that in other OS can execute right.

(Refer Slide Time: 11:47)



So, GDT is actually each entry in the GDT, so let us say this is the GDT, this is a GDT entry. So, we have so far looked at base and limit, it also has certain permissions all right, let us call it perms and which basically say you know whether this segment can be de referenced in at what privilege level right. So, you also saw that a processor the (Refer Time: 12:10) processor has this concept of rings or ring levels and you know, they are 4 ring levels, but for most practical purposes let us just assume that 2 ring levels 0 and 3 all right.

And so, this permission is going to tell you at which ring at which privilege level am I allowed to dereference through this descriptor, right. So, for example if the permission says 0, then this descriptor cannot be referenced if I am running in unprivileged mode. On the other hand, if this one says 3, then it can be accessed either in unprivileged mode or in privilege mode right.

So, apart from this base and limit descriptor also has permissions right. So, what that means is that an OS can map some of its address space always in the GDT. And so, by the way how does the processor know which privilege level I am running in?

Student: Last (Refer Time: 13:15).

Last two bits of the CS register right, so this last two bits of the CS register store what which privilege level I am running in. Also we said that a process cannot just lower its privilege level right, just I cannot just say you know if I am running at ring level 3, I cannot just say make it ring level 0, because otherwise then all protection is lost right ok.

So, if I am running at ring level 3, I can only set my segment values register values to one of the descriptors which have perms equals greater than equal to 3 right, just 3 ok. So, I cannot set CS to this descriptor. So, let us say this descriptor 10 and this is descriptor 11, you know setting it to 10 is illegal which means the two calls cause in or call the processor exception and the OS will take over right, but setting it to 11 is ok, so that allows the OS to keep itself mapped in the GDT right.

So, which means that if a processor needs to ever transition from unprivileged to privilege mode, it already has some of his address space map and so it can start using it immediately right. It does not have to set up its own address space as soon as it enters the privileged mode.

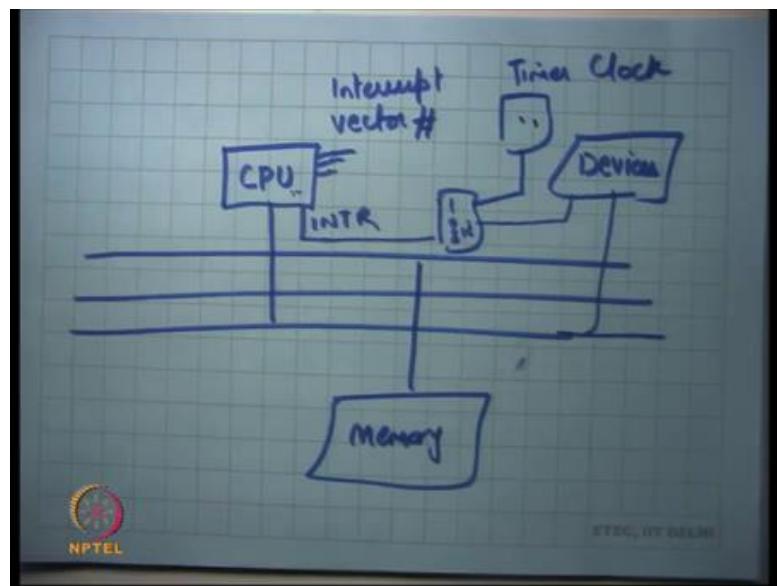
So, before we understand that let us also understand how does the; how does the processor actually change the privilege mode, change to the privilege mode right. So, I am actually executing the process and I am executing in privileged unprivileged mode and let us say an interrupt occurs right, so an interrupt occurs, and the OS needs to now run right. So, and the OS needs to run in privileged mode, because the OS will need to access its own data structures to be able to do scheduling.

For example, access all the base and limit values of different processes and then decide which one to pick and put it in there right. So, to be able to do this there is another table called the interrupt descriptor table IDT – Interrupt Descriptor Table all right, which has which is a table of you know 255 entries, each of these entries basically says has a CS colon EIP pair all right.

So, let us say CS2: EIP2 and so on and the 0, this is 1, this is 2 or let us say this is 10, 11, 12 just to just arbitrarily choosing these values right. What this means is that if an interrupt occurs; if interrupts number 10 occurs, then switch the instruction pointer and the code segment to this these values stored in the descriptor right.

So, for example if I say that a timer interrupt is going to occur at interrupt number 12, then whenever a timer interrupt occurs the execution is going to switch the code segment and the program counter EIP to these values.

(Refer Slide Time: 17:00)



So, once again let me just let me just say what I am what this means. So, let us just draw the hardware diagram. So, let us this is the CPU, and this is memory, and these are devices. The CPU has a pin which is called the interrupt pin all right and that interrupt pin is connected to some logic all right, and then that logic and that is logic just multiplexes much lots of devices and allows them to set interrupt the CPU right.

So, let us say there is a disk device as keyboard, this mouse as network, there is there and there are other devices on the motherboard and they are all going through some logic and ultimately they are connected they also have a connection to the interrupt port of the CPU right. So, when this line gets set the CPU gets interrupted right. So, whatever its was doing as a regular execution, it gets interrupted.

What does it mean for a CPU to get interrupted, it basically means started switches execution to some based on this interrupt descriptor table in this way all right? So, if so, apart from the interrupt pin there is also you know, they are also pins which specify the interrupt vector number.

So, let us say you know a device once attention it says I want to assert the interrupt pin, the interrupt gets raised with the CPU, but before it raises the interrupt pin the interrupt vector number has been set appropriately all right. The interrupt vector number determines which of these entries gets activated all right, let us say entry number 12 gets activated which basically means that the execution should get interrupted which means, whatever it was doing should get interrupted.

And the next instruction that should execute should be at this address CS3: EIP3 all right, so that is a semantics of the CPU. Yes.

Student: Then who sends the interrupt vector number?

Who sends the interrupt vector number, the hardware device all right? So, you have programmed so there is this logic that is you know sometimes programmable often programmable, you can say that you know this device has been connected to this interrupt number, this device has been connected to this interrupt number and so on. And so, the OS is can also program this device and now when the device actually asserts an interrupt that particular vector number gets sent to the CPU all right.

Depending on the interrupt number the corresponding program counter gets set right, this program counter is also called the handler of that interrupt all right. So, in other words when an interrupt occurs the handler gets called. So, now what happens is what the OS will typically do is that it will install these handlers a priori. So, for example it will say that on a timer interrupt execute this code that code will probably live here in the address space of?

Student: OS.

Of the OS right. So, what which means that the CS3 will be a segment selector of the OS I mean the descriptor that will point to will be a descriptor of the OS all right. And so, and this descriptor is allowed to have the last two bits as 0 which means ring level 0 all right, so that is one way then OS can actually rest control from of the CPU from the running application.

So, each time and so or the OS will typically install a timer handler, a timer interrupt handler. The timer interrupt handlers code will actually live in the OS address space, it will set up the interrupt descriptor table in such a way that the pointer the handler of the timer interrupt points to the handler in the OS address space appropriately. And irrespective of what I am executing as soon as the timer interrupt occurs, the process is going to switch to this code segment which may which is most likely a privilege code segment and it will start executing in OS mode right.

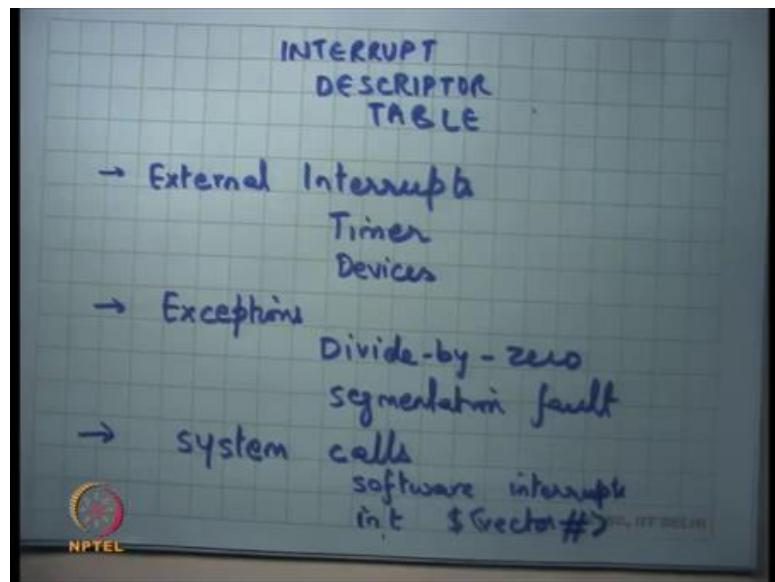
Here is an example, where CPU switches from unprivileged mode to privileged mode or ring level 3 to ring level 0. I said that a processor a process cannot just lower its ring level, but ring level can be lowered by other events like an external event like an interrupt got set right. So, in this hardware figure one of the devices will let us say a timer clock all right.

And the CPU can program the timer clock to say, I want an interrupt every 100 milliseconds right. So, the CPU has told the timer clock I want an interrupt every 100 milliseconds, because I want to you know execute after every 100 milliseconds to take stock of the situation; who all are running, who all are waiting, who should run next, etcetera, etcetera right this is called scheduling right.

So, this the OS will program the timer clock, how can it program the timer clock using the IO address space that we have already talked about right; either in out instructions or memory mapped IO, where you can just write two physical address space and you can get to (Refer Time: 22:41). You program the timer clock to in generate and interrupt every time quantum, let us say the time quantum is 100 milliseconds that interrupts gets in rated after that kind every time quantum and the OS gets to run at every time quantum right.

The OS gets to run in privileged mode obviously all right, the OS when it is running wants to be in privileged mode, so that it can do whatever it likes right. Irrespective of whether the processor was running in unprivileged mode or privileged mode as soon as the timer interrupt will occur, you will start running in privileged mode and you will get to do whatever you like to do all right.

(Refer Slide Time: 23:26)



Similarly, so we have looked at the interrupt descriptor table notice that the code segment in the CS value, the selector value of the code segment in the interrupt descriptor table has to point to one of the valid entries in the global descriptor table all right. And because of this perms field in the global descriptors, I am able to do this right. So, for this to be a valid value, I mean this selector is going to point somewhere in the table right.

And so there has to be an entry for the OS address space in this table, but I also have to make sure that the process itself is not able to change its entry to that descriptor right and so the perms field in the descriptor allows you to do this differentiation right. So, a perm 0 field can only be accessed if you know if you are executing in ring level 0 and you can only execute in ring level 0 if you for example, get an interrupt all right.

So, we saw the interrupt descriptor table. The interrupt descriptor table is so we have seen it, we seen that its used for external interrupts in particular we saw how it could be

used for the timer device all right, but it could be used for any other device as well as for example, disk, network, printer whatever all right.

So, let us say you know your program is executing the printer has finished some job and once you just interrupt the CPU, it will send an interrupt the OS if it's same OS would most likely set up the handler to live in the OS address space, because its only the OS which is allowed to talk to the; talk to the printer right the device the processes are only allowed to make system calls to the OS. So, the OS the printer handler is going to get called and the OS can do appropriate handling for the printer.

The handling way mean just say you know just make (Refer Time: 25:40) some internal data structures and then return control back to the process all right or the handling could mean you know, notify the process that something is happened. What are some ways of notification notifying the process?

Student: Signals.

Signals all right, so that is one way of let us say notifying process all right. So, other devices it can also be it is also used for exceptions all right, so for example divide by zero all right. So, what happens if a process is running in unprivileged mode, but it makes it calls an instruction which actually it amounts to a division by 0; it executes an instruction which amounts to a division by 0.

The processor is does not know what to do, because the division by 0 is undefined. So, it is going to raise what is called an exception; raising an exception is internal to the CPU as opposed to the previous example, where we saw that an external device actually asserting the interrupt pin raising of an exception is internal to the CPU all right.

But the effect it has is roughly similar, which means you know every exception is allocated a particular number. So, let us say you know the divide by zero is allocated number 9 just making it up, it is not the real number, but let us say its allocated number 9. So, the 9th entry in the interrupt descriptor table should point to the handler of a division by 0 exception right.

So, typically what will happen is if a process executes divide by zero and exception is going to get generated, the exception handler must have been set up by the OS in such a

way that. The handler actually points to OS address space and the OS is divided by zero handler gets called in privileged mode. The OS is divided by handler can do multiple things, it can either just straight away kill the process saying that you know he was not allowed to do that or alternatively he can send a signal to the process saying you had you did a floating point exception right.

So, we also saw SIGFPE in Unix which was just doing a signal to the process right. Similarly, if there is a segmentation fault what is a segmentation fault, there is a memory instruction where the address is not legal right. So, in this example all addresses between 0 and max are legal.

So, if he if the address that he tried was trying to dereference is between 0 and max its legal, but if it is he tries to dereference a negative address or he tries or you know anything if it is an unsigned, then the only possibility is if it tries to dereference an address which is greater than max, then the processor is going to raise an exception right.

How does the process know that it is an exceptional condition? The segmentation logic has this assert right, so that assert is going to fire saying that look you are less than you are you actually been trying to be greater than limit right. So, the processor is going to say there is something wrong it is going to throw an exception, once again that exception will have a certain number depending on the type of that exception.

So, let us say the segmentation exception number is let us say 14. So, the 14th entry should be set up by the OS at that it points to the segmentation handler for handler and the segmentation fault handler may choose to either kill the process or he may want to do something else or he may actually send a signal to the process itself like the SIGFPE that we have seen on Unix all right ok, so that is those are two uses of interrupt descriptor table.

The third use is actually the same thing is actually also used often for implementing system calls right. So, we saw that system calls are basically functions that are implemented by the operating system and called by the process, but clearly the system called function, the system call itself has to execute in privilege mode right, it cannot execute into unprivileged mode right.

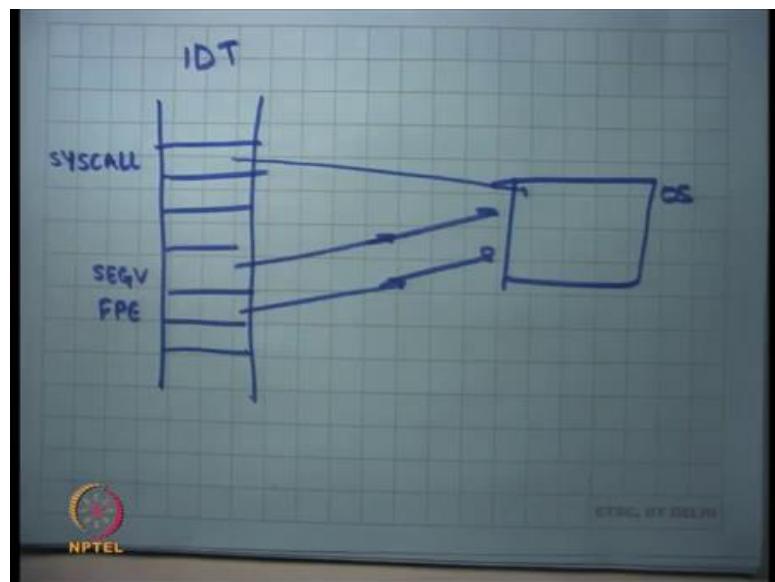
If I for example, make a read system call and the read and the file descriptor is actually pointing to a device, then I need to talk to the device right and that talking to the device may need privilege. And so, the system call itself needs to be implemented at privileged level, whereas the caller is in unprivileged level. So, the caller is unprivileged the callee is privileged, and so I cannot just implement it as a regular function call as you have seen last time right. So, one way to implement system calls or what is called software interrupts all right. So, software interrupts are nothing but instructions of the form int dollar some vector number right. So, this int is the opcode of the instruction says interrupt.

So, apart from this which is you know somebody from outside is exerting the interrupt pin this which is I made an exception like it divide by zero or a segmentation what or others. Here is an example where the software deliberately and explicitly says raise this exception all right. So, here is an instruction that just emulates the raising of the exception number of the vector number specified as its operand right. So, I can say raise, the raise exception number?

Student: Two.

70, 80 right. And so, the 80th entry in the interrupt descriptor table will get activated and that particular handler is going to get called right.

(Refer Slide Time: 31:42)



So, how will system calls get implemented in this way let us say this is the interrupt descriptor table, and you know here all these entries. Let us say you know some of these entries are pointing to let us say this is for the segmentation fault and let us say this is for the floating-point exception. And they are pointing to appropriate handlers in the OS space right. And then you know there is this special number which the operating system can define as a system call number.

So, let us say this is the sys call number which will also point into the OS address space right. So, basically one way to implement system calls is that the process sets up the arguments somewhere let us say it sets up the arguments on in the registers, and then execute this instruction.

This instruction is going to execute a routine inside the OS and that that routine should assume that the application is trying to make a system call right. And so, what does that routine is going to do is it is going to look in the value in the registers for the operands and exactly what you want to do.

So, for example, one of the operands could just be what system call you want to call whether it is fork or exec or read or write or open that could also be specified in the register right. You set up a register saying this is a systemic call I want to call, and then I just execute this instruction int that number, the OS handler gets to run, it checks the value of the register, and based on that it executes the corresponding functionality.

Student: (Refer Tim: 33:29).

Yes. Why do we need software interrupts, why do we or I think the question is also why do we need system calls? Why do we need system calls? Well, because the process cannot do everything itself; it needs some things that only the OS can do for it right that is the OS abstraction right. For example, I want to fork a process right. So, I need to tell the OS that please fork a process for me.

So, OS has an intermediate agent that is working on your behalf. You only make request to the OS that look, this is please do this for me. And the OS checks whether you are allowed to do this or not, and then depending on what he finds he is either going to do it for you or he is going to say no, I am not going to do it right. So, this there is a separation

of privilege and the system calls are a way of actually bridging that gap between tool privilege levels right.

And the question really is how you implement the system call. So, we have already seen why system calls are needed. We are really talking about why, how you implement the system call right? Well, the system call also needs, so there is some hardware support in this in the mechanisms that has described so far. The hardware support is that there is an interrupt descriptor table which has these pointers use which are which are these long pointer, long jump pointers which are which has a computer code segment and a program counter.

And then there is a global descriptor table which uses those selectors to basically say exactly which address you should go to. And then the application just similar it is an interrupt and tells the OS that this is what I want to do in some sense. So, it is basically overloading the same interrupt mechanism to also do system calls and also do exceptions right. So, this is just this is one x mechanism that can do all these common things for you.

Student: Sir, all around the other system calls also in the interrupt descriptor table like fork and all.

Are all the other system calls also in the interrupt descriptor table? Actually, there is just you know you only need one vector number for the entire system call space that you have right. You can specify what system call you want in the register. You said a certain resistor value and that is going to tell you whether you wanted fork or exec or whatever right. Because if there are 300 system calls and I am going to exhaust my id space, you just have one entry for this system call and then you specify it as an argument what system call you want.

Student: All interrupt handlers are asynchronous?

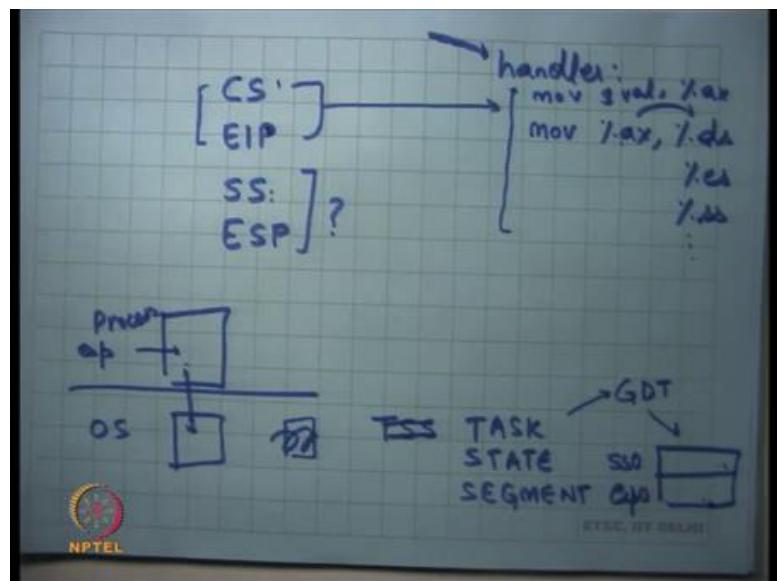
All interrupt handlers are asynchronous, what does that mean?

Student: Means when interrupt handler is called for this is running and interrupt handler is running separately.

No. So, good. So, now the question is how the interrupt handler executes, and in what environment does it execute. Firstly, the interrupt handler execution, so I was executing a certain instruction in a certain address space, an interrupt occurred or you know a soft interrupt occurred or an exception occurred, all these three are the same, I am just going to call it the interrupt occurred right.

So, I am executing a certain instruction in a certain environment and then interrupt occurred, I just switch my CS and EIP nothing else changes. So, I am still executing in the context of the original process in the same address space right, that is not completely true.

(Refer Slide Time: 36:56)



It also changes two more things sometimes which are, so it changes CS, EIP and it also changes SS and ESP in a conditional manner ok. So, on an interrupt four things get changed potentially, one is the code segment and EIP will definitely get change because you point want to point to the new handler. And what is also needed as I am going to discuss next is basically change the stack pointer you also want to change the stack. So, you going to operate on a new stack all right, and why that is needed etcetera I am going to discuss very soon right.

And SS:ESP basically determines the position of the stack. So, an interrupt can potentially also change these values, apart from that it changes nothing. So, the same address space, same everything. Of course, the handler the first thing it may want to do is

you know reload its segment registers. So, the first instruction for example in the handler could be move some value to the ds register, so that in the next memory accesses which go through ds actually are OS values ok.

Student: Sir, is this stack called convention?

This so, the new stack that comes in is called kernel stack, but you know let us just hold on that one that thought for me ok. So, let us see, so a CS and EIP handler is, so let us say this is a handler, and let us look at a typical code for a handler right. Firstly, we have established that the handler lives in the OS address space. And the CA and the descriptor for to hcs points should already mapped in the gdt right.

The first thing the handler may want to do is let us say set setup ds right, some move, something some value to ax, and then move that value to ds right. That may be the first thing I want the OS may want to do because you know very soon I am going to start executing code which is going to do memory accesses and I want that those memory accesses should actually be for the OS address space right.

And so, for them to be for the OS address space all my segments should actually be pointing to the OS is segment descriptor right. So, I may want to do it for other segment descriptors also for example, and so on right. I could also do it for let us say the stack segment right, but there is a chicken and egg problem here right.

So, let us say I am a process and here is the OS, and I executed and, in some interrupt, occurred let us say the software interrupt hardware interrupt does not matter and the OS gets through run. The OS before it starts running the first instruction needs to save some context of where it was when it was interrupted, because you know when it is going to return it needs to restore that context. In particular what context as I need to save?

Student: Stack pointer, stack pointer.

The stack pointer ok.

Student: (Refer Time: 40:28) instruction.

The instruction pointer, the old instruction, anything that it is overwriting it needs to save the old value right, because it needs to restore the old value. So far, we have only seen

that it overrides these two registers. So, those are the two resistors that it needs to save ok. So, it is overwriting CS and EIP. After it is done executing, it will want to restore CS and EIP to its original value, so that it can continue from where it is left right. So, we need to say that somewhere. The question is where does it save it?

Student: Sir.

Yes.

Student: So, handler is making the change to register to be need to these to that is well.

If the handler is making the change in the resistor it is happening in software. So, if the software writer is actually making a change to the resistor, he should be making; he should be making cop, he should save that in software, but the hardware does not need to save it ok. So, for example, the handler is overwriting ds before he does that, he may want to move he may want to save ds somewhere, but that is completely a software mechanism. Whereas, the saving of CS is a hardware mechanism because CS gets overwritten by the hardware right.

So, there is something is that the hardware is doing for example, its overwriting CS and EIP, and everything else in the software is free to do whatever it likes. Now, this is the typical thing what a software will do, but it may or may not do it and depending on whether it is actually going to make a memory access or not.

Student: Sir, but when we context switch between this and processes when we store all the registers or now process?

All right ok. So, I mean you can do all that. So, you have to save all the registers of the process that is true, but all that can be done in software all right. So, the handler itself will have logic to actually do this saving and restoring, but we are just talking about you know the process by which the handler gets called, even that process is actually clobbering some registers, and those registers have to be saved by hardware, because by the time the software gets run it will have no idea what the old values were right. So, some, some values need to be saved by the hardware.

And this in this case that we can clearly see the old CS and EIP need to be saved by the hardware right. Now, question is where does it save it right? So, one typical place where

you usually store these things is the stack right. So, you know the process may have some value of ESP, and one response to the question could be, let us just push CS and EIP to the user stack right.

So, just encrypt just decrement ESP and stuff CS and EIP on the stack, on the top of the stack. But the process is untrusted you know a process could actually set up a ESP to zero and then call this instruction called int something. And the OS is going to now try to or that hardware actually is going to try to stuff CS and EIP at address 0, where that nothing lives or some other invalid address right.

And so what is going to happen is the process is going to the processor is going to go into recursive faults exceptions right, because it tried the hardware tried to push something to an invalid address and that is going to cause another exception. And now the exception handler is going to try to execute, and again you know you are going to try to for that to execute again going to try to push the exception, the exception handler is CS and EIP into the stack which is again, so the stack is invalid you know you are basically halted the system completely right.

The way you wanted to design a system is that these processes should not be trusted right. A process should not be able to bring down the system definitely not. So, I cannot trust any pointer that the process provides me. ESP is just a pointer that the process is providing me right, and it is supposed to hold the processes tagged, but if the process is malicious, it can just set it to something wrong. Or if you if it is just you know if it is also possible that the process actually running out of stack and in which case also, I do not want to be held responsible for that.

So, what the OS needs is it probably needs a stack of its own right. So, it needs some state space which is going to say it is this is my space, and this is where I am going to save these things before I execute right, and that is where I am going to restore things from when I execute ok. So, another, so the processor provides another data structure or hardware structure to do that which is called the task state segment all right.

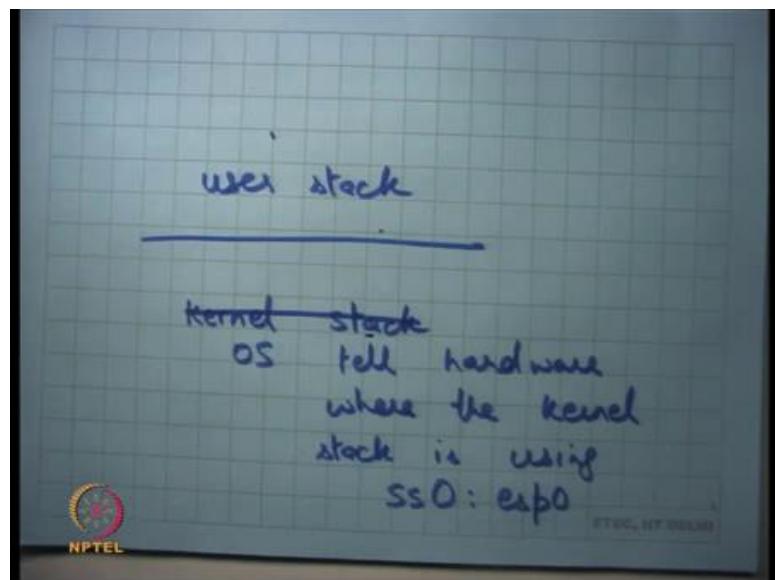
The task state segment you know is actually also a selector which points to the GDT, but eventually it points to a data structure which contains two values SS 0 and ESP 0 ok. And what this means is that anytime you switch from privilege level higher than 0 to

privilege level 0 load SS and ESP with these values before pushing the old values on the stack.

Student: (Refer Time: 46:02).

Right. So, let me repeat. Anytime I am going to switch from process to OS, before I attempt to push anything to stack, I am going to overwrite my SS and ESP with these values right. So, the x86 structure allows you to space you know there is some memory structure that allows you to specify that this is SS0 and this is ESP 0. And each time you transition privilege level from unprivileged to privilege SS will get loaded with SS 0, ESP will get loaded with ESP 0 right. And then the old values of SS and ESP, and SS and EIP will get pushed on this new stack all right.

(Refer Slide Time: 46:56)



In other words, let us simplify it. There are two stacks; there is a kernel stack and there is a user stack. The user stack is untrusted. When I make a system call and most likely I am basically executing the stack pointers pointing to the user stack or it may point to something else also, I do not care. Before I execute even the first instruction, or before I even try to push anything on the stack, I should switch stacks to the kernel stack right. We are clear on that.

The question is this operation of actually pushing has to be done by the hardware. So, the hardware needs to know where the kernel stack lives right. So, the hardware need. So,

the hardware provides the data structure, where the OS can setup values set to specify that this is where the kernel stack is.

And so, if you ever transition load this kernel stack before trying to push values on it right. So, the OS tells hardware where the kernel stack is using SS 0 and ESP 0 pointers a value all right. So, which segment what offset? Once again SS 0 should have been the value of SS 0 should be de referenceable inside the GDT right.

So, using this the kernel tells the hardware that this is my kernel stack, right now I am executing in user stack, but if there is for any reason a switch to privileged mode, the first thing you should do is load the stack. The second thing you should do is push CS and EIP of the user into this stack and not the user stack. And also push the old user's SS and esp onto this stack, because you are also overwriting the stack, so now that needs to get pushed now.

Student: But sir till that time that previous stack also will be lost (Refer Time: 49:13).

Right, I mean the hardware has some temporary buffer to basically you know make sure that it is not lost, so that is all hardware implementation right. We just have to look at the semantics of the hardware. But the semantics of the hardware is on an interrupt or on a switch, it is going to switch the stack, it is going to save the old CS and EIP, and it's going to save the old SS and esp.

So, save the old instruction pointer and save the old stack pointer and now it can run right, because this value of SS0 and ESP 0 was set by the hard by the OS itself, it is a trusted value, I can trust it right. No, process can modify it. Agreed?

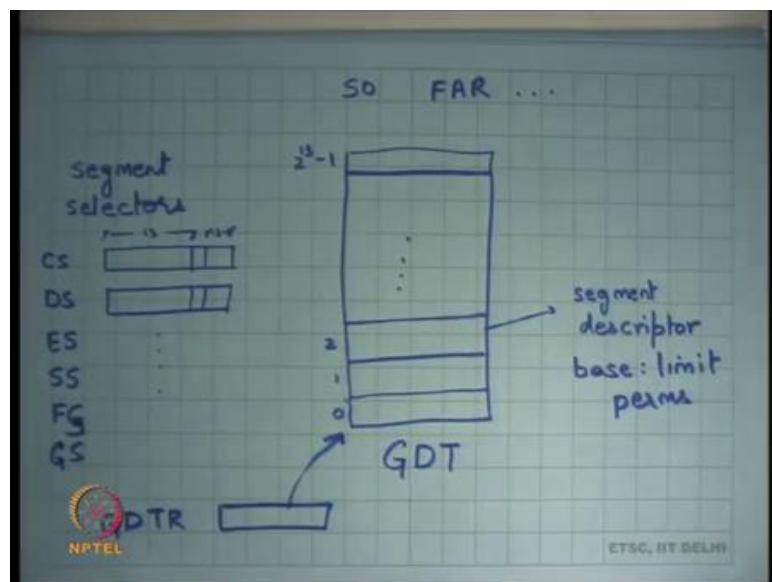
Let us stop here.

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

Lecture - 08
Traps, Trap Handlers

So, welcome to Operating Systems, lecture 8, right. So far, we have been looking at how an operating system implements the abstractions that it does. And the first abstraction we looked at was the address space, and we said look segmentation is one way to implement address spaces.

(Refer Slide Time: 00:46)



And there is something, there is a structure called a global descriptor table which lives in the physical memory. It is roughly of the size 2^{13} which is perhaps the reason why it should live in the physical memory, right because the chip will not have that much capacity to be able to store large structures like this. So, structures like this which are relatively large need to be stored somewhere else and the typical places are stored in memory, right.

And then, but on the processor you have this register which is called the global descriptor table register which points to this GDT and that is how the hardware knows where to look for when it is actually trying to execute the MMU operations, right.

Now, these segments, these registers code segment, data segment, etcetera these are called the segment registers and within them they store the segment selectors, right. Depending on the instruction a virtual address will choose one of these registers. For example, if its if you are dereferencing the instruction pointer you will go through the CS register.

If you are dereferencing you know any regular data the default segment will be the data segment, so you will dereference a DS register. If you are dereferencing the stack through esp or ebp point registers, then the default segment will be SS registers. So, there are certain default segments.

And then you can also override the default segments by explicitly specifying that this is the segment register that I want to use for this particular address. In any case, the segment selector is used to index into the GDT. So, the algorithm inside the hardware is that will first dereference GDTR, it will first read GDTR to find the address of GDT.

You know, add the selector to that value to understand where exactly the descriptor lives, read the descriptor, get the corresponding base and limit values, perform the appropriate operation of $pa = va + base$ and checking a $va < limit$. And if these checks succeed it actually uses the computed physical address to index the physical memory, all right. Question.

Student: Sir, is the value of GDTR checked that every memory access instruction?

Is the value of GDTR read at every memory access instruction? That is a great question. In other words, for every memory access do I make another memory access, so does do as every memory access that a program makes does the hardware need to make two memory access, one to the GDT and then one to the real physical address.

Well, logically speaking yes, but actually no because you know these entries get cached inside the chip, right. So, there you know there are semantics on you know when the cache gets, when the caching takes place and when it gets invalidated and let us ignore that discussion for now.

But you know you can imagine that you know the descriptors for all the 6 segments that are present on the CPU they just get cached inside the CPU. So, you do not have to go

over the bus every time to access the GDT descriptor, right. So, that is in it; that is an optimization, but let us look at the semantics for now.

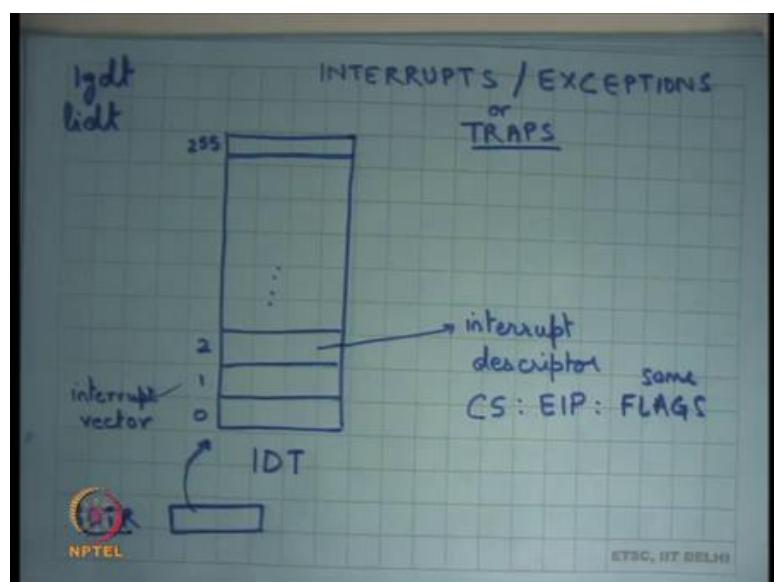
So, each segment descriptor has a base and limit and it also has permissions which basically says at which privilege level am I allowed to go through this segment, all right. So, the privilege level is determined by the lowest two bits of the CS register.

So, if the lowest two bits of the CS register are 0 which means I am executing in privileged mode, I can you know I can access any of the configured segment descriptors here or I can dereference to any other segment descriptors. On the other hand, if its 3 then I can only dereference segment descriptors that have permission set to 3, right or unprivileged, all right. So, all right so, far so good.

Basically, this what this allows you to do is every process will have a private address space, nobody else can touch it. The OS will have its own protected address space no process can touch it, and moreover each process has a uniform address space you know starting at 0 for example, right.

So, you do it the process the compiler or the linker does not need to be worried about where the process will actually get loaded. So, the loader and the linker can become completely independent in some sense, all right.

(Refer Slide Time: 05:00)



The next thing we said was look that is fine a process; this is how operating system implements address spaces for processes. But a process needs to do more in particular there needs to be a way for a process to make a system call, there needs to be a way for the OS to actually take control away from the process, on some external event like interrupt from an external device or an interrupt from a timer device for example, especially because I need to implement protection.

So, one process should not be able to run away with the CPU. So, I should be able to a run after every predefined time interval. Moreover, I should be able to get control if the process performs any illegal action like divide by 0, segmentation violation. So, the word segmentation fault has actually you know as historical roots in the segmentation procedure.

So, if a segmentation fault means that you violated the segmentation rules, right. So, you actually try to exceed the limit and so, you say violated the segmentation rules and that is how you know that is why it is probably called segmentation for, all right. So, for to facilitate this and also system calls there is a mechanism called the interrupt descriptor table or IDT as I am going to call it.

And the idea is that in case of an event which involved which is either an external event that never device needs extension it asserts a interrupt pin or it is an internal in event that the application actually executed something illegal or the exceptional condition then you know the processor is going to stop execution there for the process and look into the IDT to figure out where is the handler of this particular condition, right.

The condition could be an external device asserting something, but the condition could be internal, in either case there will be a number associated with that particular condition and that is that number is called the interrupt vector. So, the interrupt vector will determine which entry in this IDT should be dereferenced and that entry is going to be used to find the program counter of the hand law, right.

So, each descriptor in the interrupt descriptor table contains a pair compute code segment and EIP which is basically a pointer to the handler of that particular condition, right. So, for example, if you know if the network device and asserted the interrupt pin and you have assigned it number 2, then you know you should have the network device handler at

this particular address or if there is segmentation violation then you should have the segmentation fault handler at this location and all.

And these handlers will typically do what? They will either you know execute the device driver logic in case it is an external interrupt or if it is an exceptional condition, they will execute the appropriate logic to deal with that exceptional condition. So, for example, if I was in segmentation fault the operating system could say they just kill the process, all right or it could convert that exception into a signal and pass it on as a signal to the Unix process, right.

And recall what a signal is. A signal is nothing but interrupting the process execution and making a call to the signal handler of the process, right. So, in some sense the abstraction of Unix signals is very much inspired from what happens at the hardware level in terms of the exceptions and interrupts, right.

So, an interrupt also causes a handler to get executed and the signal got, but in this case it is a hard it is an interrupt handler, in case of a process it was a signal handler which was the process have registered. The process was able to resist a signal handler for itself similarly the operating system should be able to register interrupt handlers or exception handlers for itself.

The mechanism to be able to register signal handlers is provided by the OS. The mechanism to register interrupt handlers or exception handlers as provided by the hardware, right ok, all right. So, we understand handlers so far.

The other important thing is that the code segment could actually be a privileged code segment. So, even if I am executing in unprivileged mode, if this code segment has last two bits set to privileged which is 0 then you know when the interrupt is going to occur it is going to start executing in privileged mode and that is required because you want that you know whatever code you are going to have for device drivers or for exceptional conditional handling should run in privilege mode, right.

So, the hardware designers provide that facility because by allowing you to specify any CS cell, right, ok, all right. So, with that let us look at what happens if you know if. So, before I start an interrupt or an exception, so I mean I use the word interrupt for anything

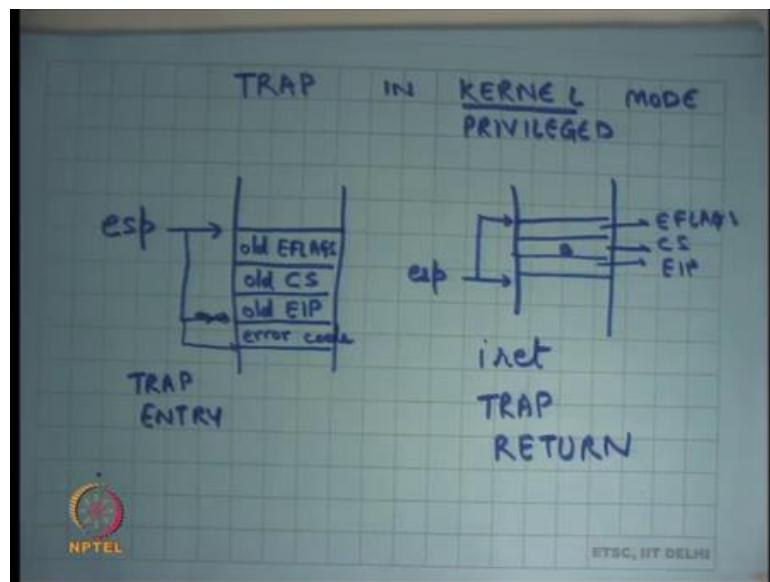
that is from an external source like external device like a timer or a disk or whatever and I use the term exceptions for something internal.

Like a segmentation fault or divide by 0 etcetera or an illegal instruction was executed etcetera, right. So, an illegal instruction is executed. One example would be a process try to execute a privilege instruction like lgdt, right. So, we saw that loading the GDTR is a privilege instruction, if you are running in unprivileged mode and the process tries to execute the privilege instruction lgdt, right.

You recall the lgdt instruction which loads the GDTR, you going to do the process is not going to be allowed to do that and what will happen is an exception will get raised and the operating system will come into action and it will decide what to do with the process, right, all right.

And another thing the IDT itself is also stored in memory. Once again IDT is a relatively large structure cannot be stored on chip completely. So, rather the IDT is stored in memory and there is a pointer called IDTR inside the chip which basically says where to look for IDT just like GDT, all right.

(Refer Slide Time: 11:14)



So, now let us say I get a trap. So, I am going to, I am going to use the word trap as a general term for interrupts or exceptions. So, I could get an interrupt I could get an exception let us just call them a trap. This is the question. Can we change the entry in the

IDT? Let us hold that thought for a moment that is a great question but let us just first understand how this works and then we will talk about security.

You know what a process can be able to do and what it should not be able to do such that you know it is not able to gain control of the system. So, let us say I get a trap in kernel mode, kernel or you know you can also say privilege mode, right. So, I am going to use privilege, unprivileged or kernel user same thing, right. So, kernel is privileged mode, kernel mode is privilege mode and user mode is unprivileged mode.

So, what happens is let us say I was executing, and a trap occurred. So, what will happen is let us say this was my stack pointer this was the value of my stack pointer. As soon as a trap occurs what I am going to do is I am going to push the old values of CS and old value of EIP and I am going to set my CS and EIP registers to the values which I get from the interrupt descriptor table and I am going to now start executing, right.

So, I got the trap in the kernel mode, I look I used the esp value at that time to push the old values on the stack and now I use I start excluding the interrupt handler, right. This is similar to how signal handling was happening, right. A signal comes you just execute that as a function call as an asynchronous function call.

Similarly, this is doing the similar thing. You know an interrupt game, or a trap happened you just executed the handler as an asynchronous function call using the same stack that the kernel was using, right ok. Another matter of detail apart from CS and EIP it also stores old EFLAGS, right. So, the interrupt descriptor table entry could also potentially contain flags resistor or certain flags and when the interrupt gets actually starts to run the semantics are that you are going to replace not just CS and EIP, you are also going to replace certain flags, some flags, fine.

So, before you start execution of the interrupt handler at this point, we are also going to replace some flags with the values that are present in a descriptor. And because you are going to change things you also need to save them, and once again you save them on the stack, all right. So, this is clear.

Student: Sir, this is stored in the kernel stack?

This is so, we are talking about a trap received in kernel mode. So, let us say does this is an interrupt while I was executing in privileged mode, this is what will happen. So, it will just push things on whatever the value of esp is currently, ok. Don't we require storing general purpose registers?

Yes, we do require saving more things, but this is the minimum that the hardware saves for you and now the control has been transferred to the handler and most likely if the first thing the handler is going to do is save the rest of the state. It may you.

Student: Sir, is a part of OS only?

The handler is yes part of the OS.

Student: But this transfer take place by a hardware.

By a hardware, yes. See basically the hardware has to change the values of some registers to be able to execute the transfer and so, whichever registers it changes, whichever is the minimum set of registers it needs to change those are the set of registers it saves. Other things it does not need to change. It is really up to the software item whether he needs to save them or not, right.

These kinds of things the software writer would not have been able to save, right because you have to say where to execute the handler and so, before that I mean it is a chicken and egg problem. So, the hardware has to come in into the middle and says, these are the things I will save and the rest of things you can say, right.

So, there are certain flags I am going to look see later which need to be saved by hardware. So, cannot E, why cannot EFLAGS we also saved in software, right. So, there is a reason for that, and we are going to discuss that later. It is just, let us just leave it for now, all right.

So, this is what happens on a trap entry, right, right, and then you know the handler is going to execute. Typically, the handler is going to execute on the same stacks. So, it is not going to change the stack value, right. So, just like an asynchronous function call, you just execute the handler and then you want to return from the handler, right.

In case of a signal, when we saw for Unix I just executed the return instruction and I will get back to where I was, right because the semantics of the signal entry was that the OS used to push the return address on the stack. And so, when the signal handler used to call return, I used to pop the return address on the stack and get back to where I was, right.

In this case, also it is going to be similar except that it is not a regular function call, so I cannot just use return to return, because return is only the semantics of with the return instruction is that I just pop the stack and look at the EIP and that is it, right. I just pop the EIP and said that. It has nothing to do with CS and EFLAGS, recall it the return instruction.

So, we need a special instruction call interrupt return iret, in trap return, you want to call it trap return, all right. And here there is if the esp is pointing at this place and interrupt return is going to pop the stack and put this into put the first value into EIP register, the second value into CS register and the third value into EFLAGS register, right.

So, this is the semantics of the iret instruction. What is going to do is its going to look at the current stack pointer and pop these, pop the first 3 words on the stack and fill these registers with those values, all right. So, this is interrupt from, return from interrupt.

Student: Sir.

Yes.

Student: Yes, (Refer Time: 17:51).

Yes. So, right now I am talking about a trap in the kernel mode, right. So, because I was executing in the kernel mode already.

Student: esp is of kernel.

So, esp is of the kernel, all right. So, I am already done executing in the kernel mode. So, esp is trusted in this case. So, I can do these things. Next, I am going to talk about traps in the user one that was the discussion that we have we were having yesterday, right.

As a matter of additional detail there are some vectors, some interrupt vectors which actually push 4 words, all right. This is just a matter of; just a matter of the fact really there is no there is no fundamental behind it, but there are certain vectors for their which

there are certain extra, this one extra value that is pushed that is the error code, right. So, for example, the page fault. For example, you can say you know a certain exceptional condition could additionally push this error code for the handler to know exactly why that exception occurred, right.

So, certain exceptions can occur because of various reasons and so an extra value is pushed on to the stack to indicate that, right. So, the only the hardware knows exactly why this exception occurred. So, one exception could actually be representing multiple conditions and so error code basically tells you which of these conditions is actually the reason for that exception.

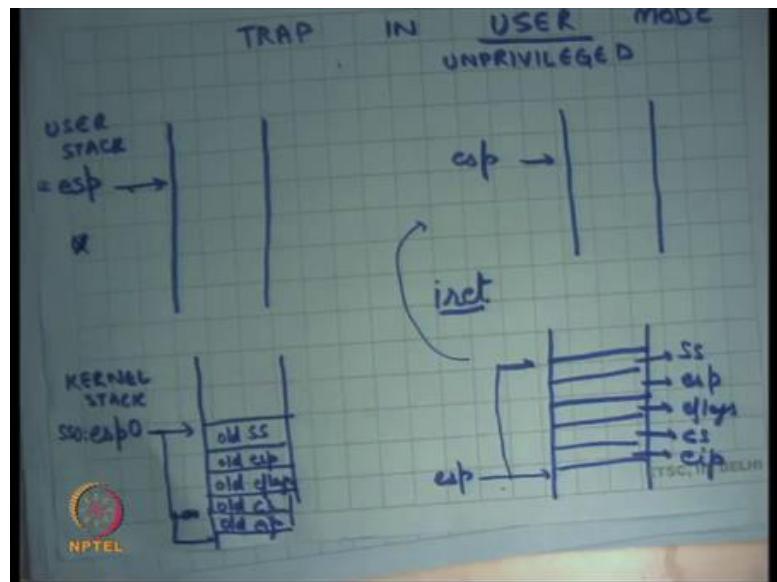
What this means is because some vectors push the error code and others do not. The handlers need to be appropriately set up such that they understand this, right. So, the handler for the vectors which push the error code will be set up to know that an error code is already there, and the handlers for which the vectors which do not push the error code to know that it is not there.

Just to simplify things typically on x86 and OS handler will all the handlers for which the error code is not pushed the handler will the first thing it will do is just push a 0 value there, so that this stack becomes uniform, right. So, it can assume a uniform stack frame really, right. So, if there was no error code pushed by that vector the OS in software will just push a 0 for that particular, in that particular handler.

Student: So?

On return the iret instruction does not assume the presence of error code, all right irrespective of what happened here. So, it is a responsibility of software to pop the error code before it causes the iret, ok, all right, ok.

(Refer Slide Time: 20:23)



Now, let us look at a trap in user mode, all right or let us say unprivileged mode, ok. Once again, I was executing and here is my stack pointer that is where it is pointing, and a trap occurred. Now, what should the kernel do? Can it do the same thing? Can it just push; so, clearly it needs to override certain registers for example, it needs to override CS and EIP definitely, it also needs to be override EFLAGS, ok.

So, now question is where it pushes the old values of these registers because it needs, they might interrupt return. So, where does it push them? It wants to push them on stack. But can I just push them on this stack?

Student: No, sir.

Why? Because this particular value of this register is modifiable by the user and the user could set it to anything which cannot be trusted, right. I do not trust. So, the model is that I do not trust the process. I do not trust the process in the sense that I would not let the process bring down the whole system no matter what the process does, it should never be allowed to bring down the whole system.

In this case, if the process had just set esp to 0 for example, right or some you know some invalid address and then, right. So, let us say esp was set to 0 and then it executed some exceptional condition which caused the trap. Then what will happen is the kernel will try to push on a full stack, right, 0 cannot be decremented any further and so, it will

get into an infinite exceptional condition and this is the CPU will actually halt, right. So, that is not; that is not acceptable.

So, what I am going to, so what is really needed is that on a if you receive a trap in user mode you should also switch the stack pointer before you start pushing things, right. So, so what happens is on a trap it actually switches to another stack and that stack is let us say esp0 and then on that stack it is going to start pushing things, right. Notice that because I am modifying more registers now, unlike the previous case where I was just modifying a CS and EIP I am now also modifying esp.

In fact, I am also modifying SS, right. So, the semantics are that I am going to modify the entire virtual address which is represented by SS: esp. So, initially if it was SS: esp now it becomes SS 0 colon esp0 and so, I need to save the old values of SS and esp and I am going to do that on this stack, right. So, what I am going to do is I am going to push 5 bytes here and I am going to say let us say old SS, old esp, old EFLAGS, old CS and old eip, right

So, these are the things that the hardware saves for you and now the interrupt handler is set up to be able to run in a secure environment. So, it has a secure stack that I did not trust, and it has it is on the, right instruction pointer and now it can start running and it may want to save more things. And typically, what it will also save those things on the same stack on which it was started because this is trusted stack anyways, ok.

Student: Sir, second one is a kernel stack.

The second one is the kernel stack. This was the user stack, right. Now, what happens if when the kernel executes the iret instruction? So, in the now the interrupt handler will execute, or the trap handler will execute in the on the kernel stack and eventually it will want to return. And it should actually return to exactly the same point in the user space we had actually left off, right.

So, once again it is going to call the iret instruction and the semantics of the iret instruction is going to once again, but you know let us say this was the esp at this point. It is going to pop off the first 3 words and set up set those up as CS EIP and EFLAGS just like before.

But now it is going to see that oh the CS is actually an unprivileged CS, right. So, the CS that it is actually popping into the; into the register is actually an unprivileged CS. It can see that from its value and so, it is going to realize that because it because I transition from a privileged to an unprivileged, I am transitioning back from a privileged to unprivileged mode there must have been two more words that have been pushed and so, it also pops off those two words and actually pops off 5 words to set to basically reload these registers back again, all right.

So, in the previous exam case the iret was just popping 3 values in this case iret is popping 5 values. How does iret know how many values to pop? By looking at the value that was popped in the first 3 words, right or by looking at the value of the second word, right that contains the CS, all right.

And so, when you execute the iret instruction you are actually going to get back to the user stack, right because you have changed the value of SS and esp, so you are actually now going to start executing in the user stack with exactly at exactly the same EIP at which you left off, ok. So, now let us talk about security. So, what prevents a user from being able to take control of the system?

So, we said a user can in basically sandboxed within his own address space because firstly, he cannot modify GDTR; secondly, we said the GDT itself should live in a portion of memory that is not accessible by the user, right and thirdly these the values of these selectors can only be set to one of the values that have been put in the GDT, right and the OS should be careful that it only puts the ah, right values or permissible values in the GDT, right. So, that is how I was ensuring that a user is not able to jump out of its address space question.

Student: Previous one, how do we interpret in the kernel mode that esp never becomes 0?

Good question. So, how do we ensure that the curve in the kernel mode esp never becomes 0 or esp never underflows, right or I had never done out of stack? So, this is the, this is something that an OS designer has to be careful about. Nothing in this world is, nothing is infinite, right, infinite. So, even in the user mode a stack is never infinite. It is just an illusion of infinity, right.

So, if you ever try to cross your boundaries you are going to if a process ever crosses its boundaries, it is going to get a segmentation fault, ok. And OS is should ideally never cross its boundaries, so the OS designer should write its OS in such a way that there is a bound to the maximum stack, stack length that you can have, all right. I am going to see how that is done.

So, an OS designer or OS writer or a kernel developer has an extra bound that you know you cannot grow the stack too much, there is a maximum bound to how much is stack can be. Typically, a stack would be for example, the Linux kernel has a stack of one pay of around 4000 bytes or maybe even 8 thousand bytes, right. So, between 4000 to 8000 bytes are enough in general for the stack, right.

Even your xv 6 kernel which we are going to look at an academic kernel called xv 6 (Refer Time: 28:33), it also has you know 4000, 4096 bytes of stack and that is enough for of course, ok, all right, ok. So, we saw how; this is basically saying that a program can never jump out of its address space. It is also saying that a program can never execute an instruction which is privileged because I am going to run it in the privileged mode etcetera, ok.

And the program can never lower its privilege level. So, once you have set up the privilege of the CS register, I cannot just lower their privilege level. But we also saw that the IDT is a way for the process to actually lower its privilege level, right. So, for example, I can just execute some exceptional condition and I will now be executing in privileged mode.

So, the first thing is that the OS should ensure that all these entries all the handlers of the IDT are appointing, all the entries in the IDT are pointing to valid values. If one of these IDT values is pointing to some garbage then an OS can actually cause that particular vector to get fired and you are going to actually try to execute some in invalid instruction and the system can get down, right.

So, the first thing is the IDT itself should have completely same values. The second thing is the instruction to load the IDT which is the lidt instruction, which just load the IDTR should be a privileged instruction, right a user should not be able to just say lidt and because of the user can say lidt then you can just take control of what gets executed in privilege mode. So, the instruction lidt should be a privilege instruction, all right.

The third thing is that IDT itself should live in the OS address space, it should not be able to, it should not be visible to the for any process, ok. So, only the OS can set up these values. The OS needs to be very careful about settings of these values, so the user cannot take advantage of any bugs in the OS plus you basically ensure that this structure is not modifiable by the process, ok, all right, ok.

Student: Sir.

Yes, question, ok. Is it possible that a process wants to handle an interrupt in a different way than another process? It is a matter of what abstractions the OS is providing to the process. In the abstraction that we have seen so far that is not possible. A process has no idea what an interrupt means, right. It only understands system calls and signals, ok. So, it does not make sense to say that a one process should be able to convert and to control the handler of a particular interrupt, right.

The process can control the handling of a particular signal, right. So, that is the difference. Of course, you know; so, what you are, what you are put is you put a layer in between the hardware and the process and you have said that this is these are the permitted things that you can do and I am going to implement those things, right.

And we have seen one type of abstractions that are the Unix abstractions, and then there are other types of abstractions which actually allow the thing kind of thing that you are saying, all right and they are performance advantages to be able to do that, right, but it also makes things more complex and etcetera and you are going to look at those trade-offs later on, ok, all right.

So, in this figure when I said that when our trap is received in user mode I actually switch the stack to as a 0 and esp0. There is a minor matter of detail where does the hardware or get these values, SS0 and esp0 from, right. So, there needs to be some way of telling the processor, look these are the values of SS0 and esp0, before you actually before the OS actually gets control to the user he should set up SS0 and esp0 appropriately, so that if a trap occurs while the process was executing the hardware knows that this is what you should load, right.

(Refer Slide Time: 33:01)



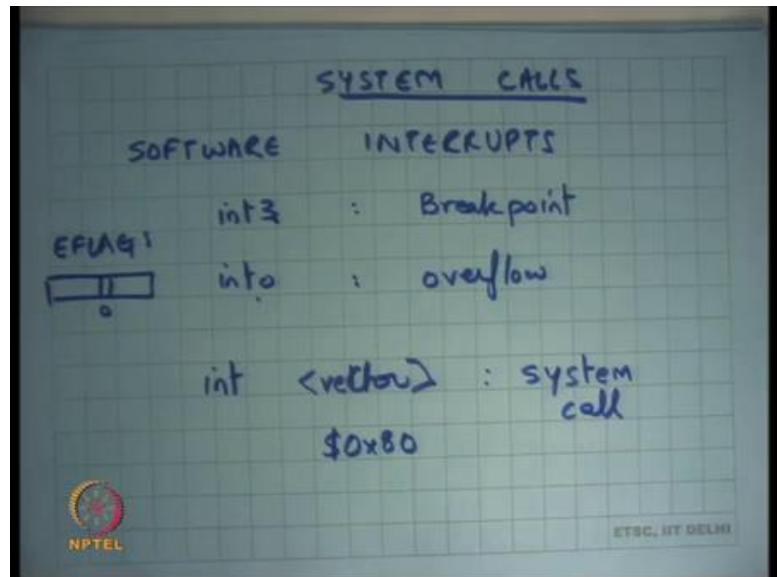
And on x86, there is a structure called task state segment which allows you to do this and they have more detail they are going to find out in your programming assignment. I am not going to go into detail, let us just for the purposes of our discussion let us just assume that there is some place where you can store these values and tell the hardware that here is where you should pick up these values from, right when you get an interrupt in user mode, ok.

Another thing I would point out is that notice that when I am talking about the semantics of instructions like iret, while the iret the semantics have been designed in such a way that they complement or bracket the operation at interrupt entry or they you know, so whatever the interrupt entry is doing iret is undoing. But actually, iret exists as a separate instruction in itself, right which has these semantics that is going to pop off the first 3 words and load them into these registers and depending on its values and maybe pop off two more words etcetera.

So, actually between the execution of the entry between the entry and the return the interrupt handler could potentially modify these values, right for whatever reason. Typically, it will not modify these values, but sometimes it may, right. One example where it may modify these values is for example, when you want to set up the first process or let us say when you implement the fork system call, right. So, what happens in a fork system call?

Before we discuss that let us also talk about how system calls are implemented. We discussed that lesson yesterday but let us just review that.

(Refer Slide Time: 34:38)



So, apart from interrupts and exceptions there are also something called software interrupts. These are interrupts that the program can actually invoke. So, instead of some exceptional condition happening or some external device saying that I want an interrupt to get handler to get executed.

The program itself could execute an instruction like int 3, it is a one-byte instruction or int o which basically says invoke this particular interrupt or trap. The semantics is that this is going to simulate an interrupt or vector 3, right. This is going to simulate an interrupt of vector whatever o stands for, overflow in this case, right the semantics for this is that this is basically used for debugging. So, this is the breakpoint, right.

So, if you have wondered how GDB works for example, one you know it basically; so, if you put a breakpoint at some point you say I want to stop the execution at this particular point, right at this particular value of the instruction pointer, what GDB does one way to do that is basically that GDB writes this particular instruction int 3 on that particular byte.

So, let us say I wanted to get interrupted at EIP 1000, what GDB going to do is it is going to write int 3 instruction at 1000. It is going to replace the original contents of that

EIP of that memory location and put int 3 there. What will happen is when then program gets executed as soon as it reaches that point it is going to execute int 3 and an interrupt going to get simulated. The interrupt handler will be the OS interrupt handler and in this case the interrupt handler will know that this is a breakpoint interrupt handler.

The breakpoint interrupt handler let us say what it does is it converts the exception into a signal that it gives to the GDB process and so, the GDB is installed a signal handler which basically says stop execution and return back to the user with a prompt and ask for the for the next command from the user, right. So, this is one way of implementing breakpoints, right. So, a good example of why software interrupts are used, ok.

Similarly, in interrupt overflow is an overflow condition. So, this basically says that if now if in the EFLAGS register the overflow bit is set then cause an interrupt. So, the way the hardware designers imagined this to be used is basically you perform some computation and then you execute the int instruction. So, if the execution actually created an overflow it will cause an interrupt. If it did not cause an overflow it will not cause an interrupt.

And so, the idea is in the common case when there was no overflow you will just you know very quickly go to the next instruction, you do not have to have if then else kind of logic in your code. So, this makes gives you a nice, very fast way of doing this kind of exceptional condition handling, all right. And of course, then there is the normal int instruction which can take any vector number, and this can basically say simulate this particular interrupt, right. And this is what we use for system calls.

So, example the particular vector number let us say you know the Linux kernel had been using the number 128 or hexadecimal 80, to do the system call. So, basically means if a process makes int dollar 0x80, it is going to simulate an interrupt at vector 0x80 and the handler at 0x80 is going to assume that a system call was made.

It is going to also assume that the arguments of the system call and the system call name itself is stored in certain places. For example, it is stored in the registers. And so, it is going to read the value of the register to figure out what the argument what system call I need to execute for example, exec fork etcetera and what are the arguments for to it, right. For example, this the address of the string for exec etcetera, ok.

So, system calls are also implemented by using this interrupt descriptor table structure and, ok. And so, and so, we were discussing how iret can be used. So, we said that iret can actually iret has us these semantics and handler may want to actually change these values before the interrupt runs and one example where you would do that is the fork system call, right. What happens on a fork system call? You make a system call; the handler is a system called handler it figures out that you are trying to call fork.

What it does is it creates a new address space and copies your address space into the new address space. One way to create a new address spaces get that much get the same amount of memory in physical memory, right, copy all the contents, set up a new base and limit in your internal data structures and set up the stack in this way such that EIP, CS, EFLAGS, ESP, SS is identical to how it was in the process which called the fork with the only difference being that the return value should be different, right. So, how does it do that? Where is the return value stored?

Student: ES.

Student: ES.

Let us say it is the return value of the system call is in the ES system, so it just changes the value of the ES register and executes the iret instruction. So, it is a set of new stacks, it copies these values, it maybe changes some register and causes iret, right.

Another way you know a fork system call could potentially return is basically you know store the value on stack, right. So, let us say the return value is coming on stack even that is possible. What the handler will need to do in the new process is dereference the stack pointer of the user and maybe change some values there before calling iret, all right. And so, the stack will actually see new values, slightly different values let us say, ok.

Similarly, you know when the kernel boots there is really no process, there is really no user mode, everything is run running in privileged mode. So, when you create to create the first process you know for example, the init process on Linux you know what the kernel will do is it will just set up the stack in a certain way, it will set up the address space in a certain way such that you know a certain executable is loaded in the address space and now we will just call the iret instruction, and so, now, the first instruction of init starts executing.

So, even though it actually never existed before the iret, iret is basically simulating as though I am returning back from a system call that made, right. So, another I mean basically a kernel can set up a fresh stack and still call iret or it can modify these values and still call iret to a to implement desired functionality.

Student: Sir, when while coating cannot we just move the values into the CS and EIP to move instruction instead of first coating with stack (Refer Time: 41:54) iret?

Cannot it just move? So, you cannot just move into CS, right, there is no instruction that says move into CS, there is no instruction which says move into EIP. There has instructions like 1 jump.

Student: L 1 jump.

Right. So, that is an interesting question. Can you just say 1 jump to this particular value and let us see why is that not allowed? So, you also need to change the stack, so you will basically need to load the stack from the user mode to the kernel mode and then you are going to call 1 jump.

Student: Sir.

I believe it is not allowed in the processor. So, you cannot just say 1 jump from one privilege level to another privilege level, right, but let me confirm that. So, exactly why the x 86 architecture does not allow you to adjust to 1 jump. But more importantly even this is an atomic operation, right. So, you can just basically set up the stack and the EIP in one go, as opposed to being it we are doing it in multiple instructions.

So, first you will have to load the stack and after you have loaded the stack you are still executing in privileged mode and so, that has also has its own security implications you know executing on an unprivileged stack in privileged mode it has its own problems.

Student: So, we want to need to agree to return to both the process (Refer Time: 43:15) parent and the child.

Right. How do we do that? That is easy. The parent can just return. So, for the parent it was just a system call, a regular system call, right. So, just how just like it returns oh my

regular system call is going to call iret and is going to return from it. For the child you need to create a fresh stack, right.

So, here is the process, it made a system call, write the system call basically internally figured out its a fork and so, it created new state and create a new stack, it added a new process to its list of processes and now it just calls iret on the original stack, right. So, the parent can just continue as it is, right. And now the child on the other hand will get will continue in on its new stack.

Student: So, we do not iret on the child.

We call iret on both, right. So, only one process enters the kernel and two processes exit the kernel.

Student: Need data.

And the stack.

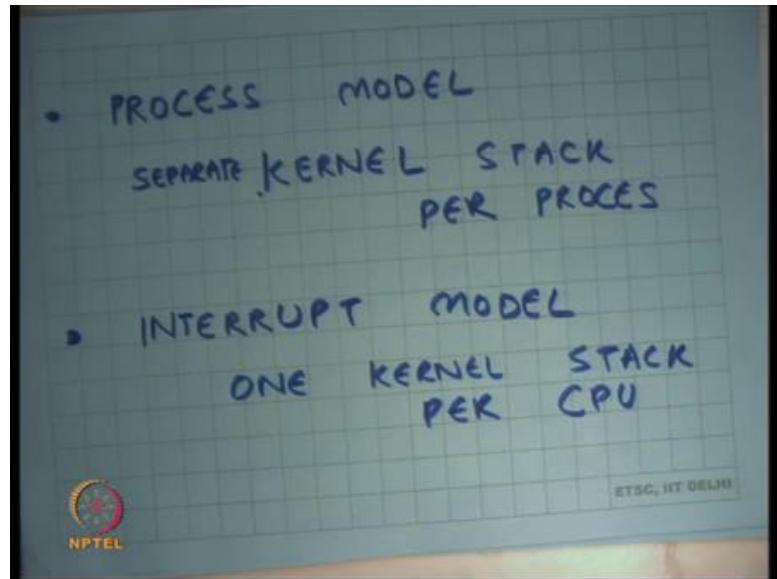
Student: In the stack.

So, we are copying the entire address space and we are setting up the stack pointer to be identical, so that basically means you are copying the stack also.

Student: But there is only one kernel stack, right.

I see. So, what basically, ok. So, now, you are now you are asking how many kernel stacks there are, all right. So, that is your question, good, ok.

(Refer Slide Time: 44:46)



So, how so, there are two ways of kernel as implemented one is the process model, where there is a kernel stack, separate kernel stack per process, all right. So, in this case, what will you do on a fork? You will create a new kernel stack and you will copy the entire kernel stack, so apart from the user address space which has nothing to do with a kernel stack you wait and so, you going to have a separate process a kernel stack per process.

So, when you fork you going to actually have two stacks and you are going to call iret on each stack independently. So, that is how you implement one entry and two exits, right. There is another model in which you can implement things which is called the interrupt model. So, this is basically you know how you implement your kernel. In the interrupt model there is just one kernel stack per CPU. So, let us say there is just one CPU in your system then there is just one kernel stack.

And so now, what the kernel needs to do is that it needs to store; so what will happen is that there is just one kernel stack and that is the value that goes into SS0 and esp0 that the hardware knows and that is permanent and so, whenever you enter you enter on that stack. But you are going to when you switch, so let us say you know when I was let us say a function process make a fork system call and I was executing on the stack and now I created a new process called child and now I wanted to switch to the child.

So, what I will do is I will save all the contents of the kernel stack in some other data structure and load the content of the child's stack from his data structure into the stack, right. So, eventually the same thing you basically need separate states in the kernel which simulates a stack per process, right. In this case, you are having you are actually having a separate kernel stack per process, in this case you actually have only one stack which is visible to the hardware, but internally you are swapping state to basically fill that stack, right.

In this case, you will actually tell the hardware, so on each context switch you are going to tell the hardware that this is the this is stack, this is the value of SS0 and esp0 you should use, right on every context switch you change the hardware structure. In this case on every context switch you do not change the hardware structure you just do this internally. One of them, we only have one CPU let us assume, and then.

Student: Sir. So, we cannot call them parallelly before the iret?

So, you cannot let us say there is just one CPU. So, what will happen is one CPU, one process made the fork call, it created a new process, it added it to the list of processes that are possible to run, right and now let us say the parent continues to run. So, parent is going to call iret and now parent can continue to run and then let us say the parent says I want to get out of the CPU or let us say a timer interrupt occurred and the OS forcibly brings him out of the CPU and now this process's turn comes and so, now, he gets to run and now he will call iret.

Student: But at that point already popped out all the as the (Refer Time: 48:03).

No, we popped out things from the parent stack. So, we have a separate stack per process, right. So, we popped out things from the parent stack the child stack still remains.

Student: Sir, in the interrupt model what it is exactly (Refer Time: 48:15) as once you have popped off the parent child?

In the interrupt model you know it is basically the same thing except I mean you basically. So, storing the value with which the stack needs to be initialized when it gets context switched.

Student: And we are storing in?

In some data structure, right. So, let us say in that process, in the list of processes you also saying you know this is a these are the values which you should initialize the stack before you start it running.

Student: (Refer Time: 48:38).

Ok.

Student: So, (Refer Time: 48:40).

So, I mean the there is no fundamental difference really, in one case you are exposing it to the hardware and the other case you are keeping things internally, all right, ok, all right. So, question.

Student: Sir, what will happen in case of (Refer Time: 48:57) parent and child?

Right. So, what will happen? So, this seems to be confusing people that what happens in case of one kernel stack. So, let us say, this is one kernel stack, all right, but each time you context switch you are going to reinitialize that kernel stack with certain values, right and so, for in case of parent and child you store that you know this is what you should be initialize it before you start running it, ok. So, that is the difference.

Student: (Refer Time: 49: 26) then where is that stored, parents where are these values that (Refer Time: 49:30) actually?

In the first, clearly in the OS address space, so where are these values stored? Clearly in the OS address space how they are stored, in some data structure which is maintaining the set of all the processes and associated information, right. In any case, when I have a separate kernel stack or process, I am also storing the stack with the process, right.

So, this is the this is your stack, this my stack, this is his stack etcetera and so, I am going to load the esp with that stack. In this case the esp remains the same I just initialize the memory locations that is the only difference, right.

Student: Sir, iret possible that we can copy all these 5 contents who some (Refer Time: 50:07) twice the copy of the same?

Is it possible to make two copies of the same thing on the same stack?

Student: Yes, sir.

I mean for intentionally or unintentionally?

Student: Intentionally.

And then why?

Student: Then, first time when we called iret for either child or parent then we get the first time (Refer Time: 50:31) second time when we call the second time.

No, it has a lot of problems. So, the question is can we use the same stack for both the parent and the child. And actually, I have two frames one for the child on the parent and use the same stack and not having two contexts switch. There are also other processes in the system, right.

So, you know I mean the cleaner way would be that you basically say that you know this is yours, this is how you should initialize things. So, other it is possible that between the parent gets to run and between the child gets to run the other process that get to run in the middle. So, you know why you would want to do that, ok, all right.

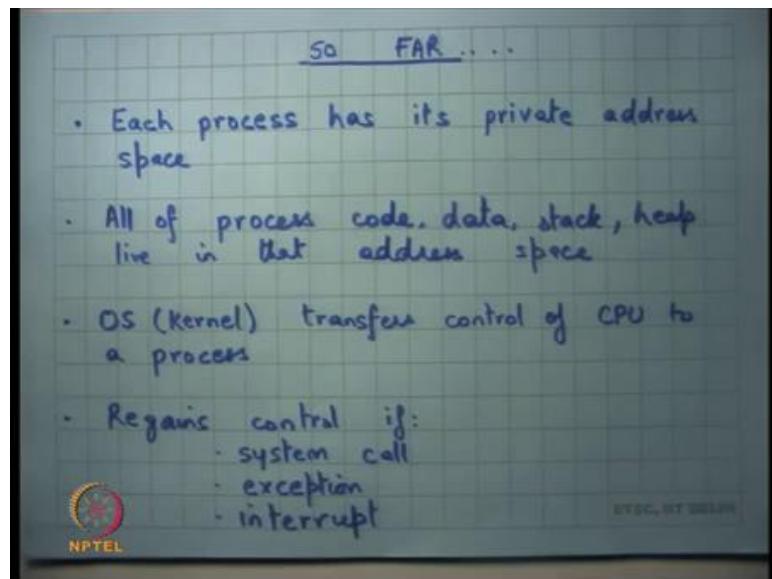
So, your programming assignment, you are now roughly ready to actually start on your programming assignment and you should start on it immediately, ok, all right, ok. So, let us stop.

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

Lecture – 09
Kernal memory data structures, Memory Management

Welcome to Operating Systems, lecture 9 right.

(Refer Slide Time: 00:29)

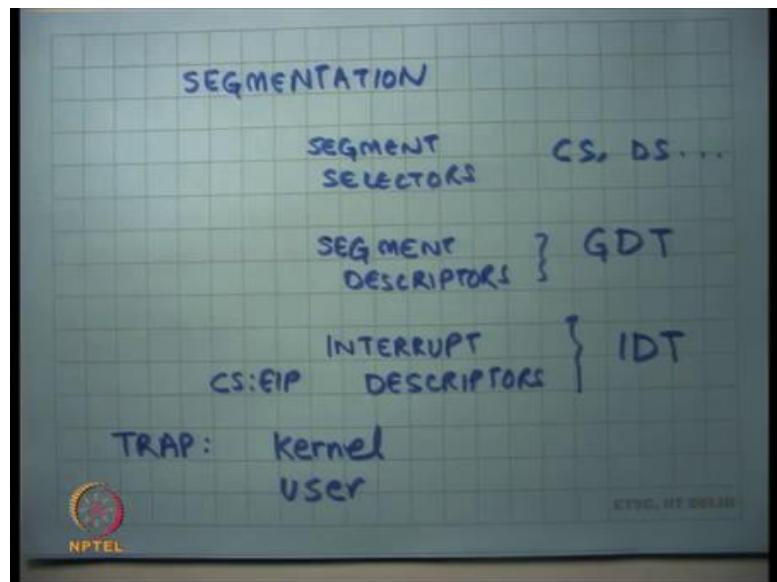


So far, we have seen operating system abstractions typical operating system abstractions, a process has its own private address space, all of the processes code, data, stack and heap live in that address space right. And, depending on what are the values of certain registers like eip, esp etcetera you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

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

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

(Refer Slide Time: 01:31)



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

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

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

And, then we said how do you transfer control? An application a process should not be just allowed to low say that now I want to execute in privileged mode right so, but you

sometimes want to execute in privileged mode. For example, if you want to make a system call so, or there is an interrupt or there is an exception etcetera.

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

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

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

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

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

So, when a if a trap occurs in the kernel mode the old CS and old EIP and old EFLAGS are stored on that stack right. On the other hand, if a trap occurs in user mode it is not safe to store these values on the user stack. So, what happens is if a trap occurs in the

user mode or the unprivileged mode the processor also switches the stacks right and where does how does it know where to pick up the stack from?

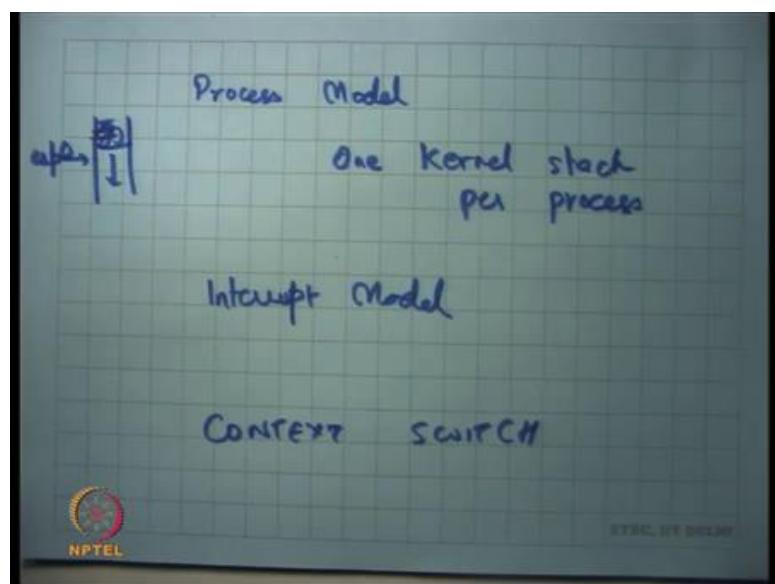
Student: (Refer Time: 05:52).

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

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

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

(Refer Slide Time: 06:55)



So, we said that the kernel could be implemented in 2 ways – one is the process model where one stack per process one kernel stack per process. In this mode whenever you transfer control to a process you have a special area which you call this process is kernel

stack right special memory area and that that is basically what you load into the task state segment. So, before you actually transfer control to process P1, you load P1's case stack; before you transfer control to P2 you load P2's case stack and so on right.

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

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

And, so in the process model for example, I could say here is my kernel here is my esp0 and I could have already pushed some things here and when the trap occurs actually the 5 values get pushed here and so, the stack is not completely empty. It also has other stuff which the handler may need right, but typically this is not used you also start to the fresh stack in the process mode right you do not really have things that are saved previously and similarly in the interrupt model.

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

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

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

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

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

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

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

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

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

In the process model you do not need to save the contents of the stack you could just save the stack pointer and that is enough right. So, that is the; that is the real difference,

there is no fundamental difference. In one case you are storing the contents of the stack if there is a context switch that occurs within the kernel execution, in the other case you are just switching the stack it itself alright.

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

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

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

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

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

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

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

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

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

Student: We can actually change further process.

Yes. A process is free to change its segment selector right. So, if the GDT has segment descriptors of other processes then a process can easily just change override a segment selector to point to somebody else's address space. And, so that is a security risk P1 can now access P2's memory which was not supposed to happen right. So, you actually need to overwrite the GDT to do with that. Sure, question.

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

Sorry.

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

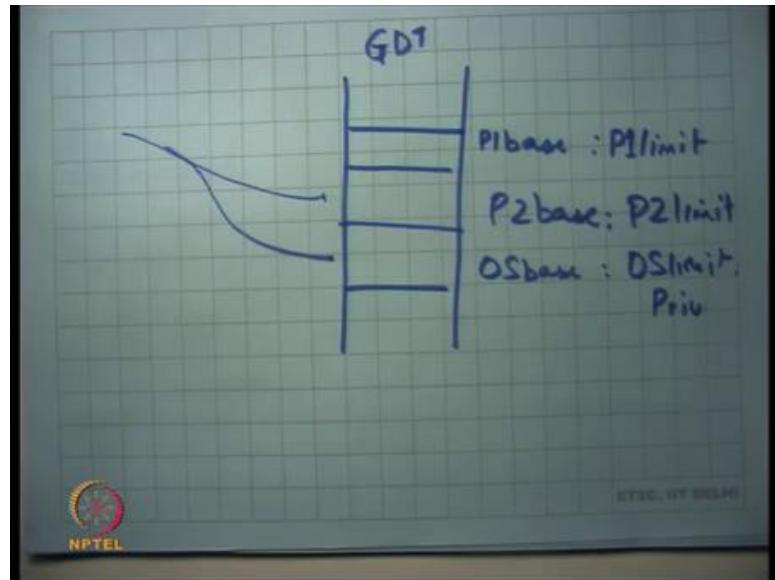
That is in the descriptor right.

Student: Yes.

So, ok.

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

(Refer Slide Time: 15:39)



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

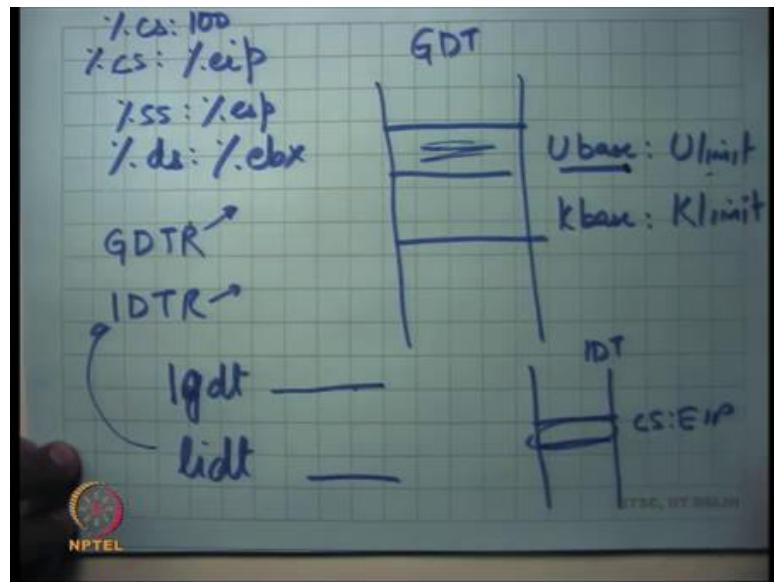
Now, when I context switch if I just switch the segment selectors you know, and a process is allowed to overwrite its own segment selectors. So, what all the processes need to do is point to this segment selector right. So, P1 can now point to P2 and so, that is a problem right. P2 cannot still point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

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

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

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

(Refer Slide Time: 17:17)



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

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

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

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

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

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

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

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

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

And, so basically what will happen is that the process which was executing will now straight away jump to the signal handler of the FPE, the floating-point exception right. And, so the signal handler of the floating-point exception can be customized according to

the process ok. So, the process can customize the handling of at the user level handling. We cannot customize the process the kernel level handling of a certain condition alright.

Student: Sir.

Yes.

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

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

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

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

Hm.

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

Ok.

Student: So, these will get executed first.

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

interrupt handler gets called, the kernel interrupt handler is the one which is actually going to generate the signal.

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

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

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

Student: Ha.

If all that an interrupt does is generate a signal.

Student: Yeah.

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

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

So, you know you could you know Unix signals as not a necessary abstraction I could just say you know I do not need you know my operating system does not support signals

you know. If you have made if you if some error condition occurs if you make an exception I will always kill you alright and if you for example, make some kind of inter process communication or some kind of you know device communication then it is a responsibility of the process to keep checking rather than a signal based mechanism.

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

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

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

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

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

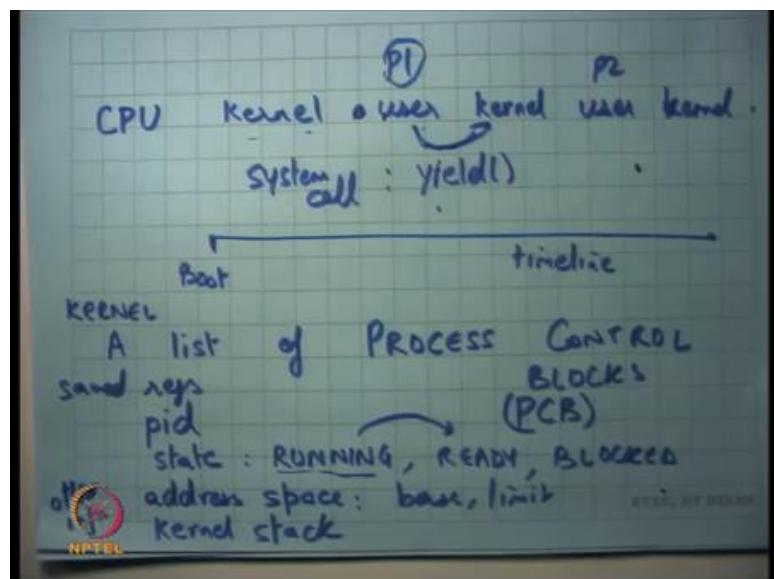
In both case the operation is the same is just the way of notification that is different alright. Finally, I want to say that there are 2 pointers on the chip. There is the global descriptor table register which is the which is points to GDT and then there is the interrupt descriptor table register.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

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

Student: Physical address.

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

(Refer Slide Time: 27:51)



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

Student: Sir.

Give me a second ok. Question?

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

How does the?

Student: (Refer Time: 28:02).

Sorry, how does the what fit in?

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

How does the physical address fit in the GDTR?

Student: (Refer Time: 28:15).

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

Student: Physical address.

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

Student: (Refer Time: 29:00).

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

Student: Virtual address, virtual.

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

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

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

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

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

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

whether it is running a process could be actually running right now right or whether it is ready to run, but not running.

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

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

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

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

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

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

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

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

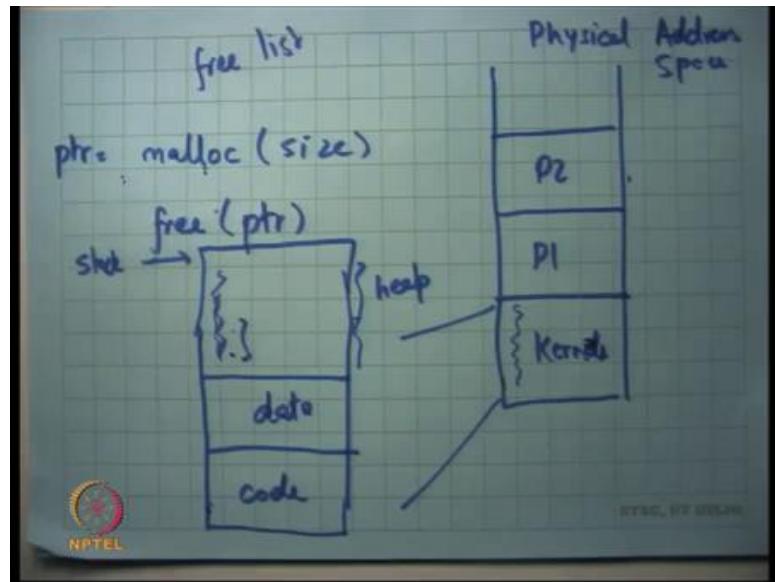
So, what do you do? So, we have discussed this before the kernel has you know has heavy unridden mechanisms in which case it basically sets up the hardware device called timer. So, every you know every computer system will have a device called the timer device on it is motherboard let us say and so,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

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

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

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

(Refer Slide Time: 37:23)



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

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

And, so what malloc takes is a size and returns a pointer and what free does is, it free is that point. I am sure all of you have used malloc and free yes alright. So, and of course, malloc can fail right. So, malloc may return the null pointer in which case the malloc

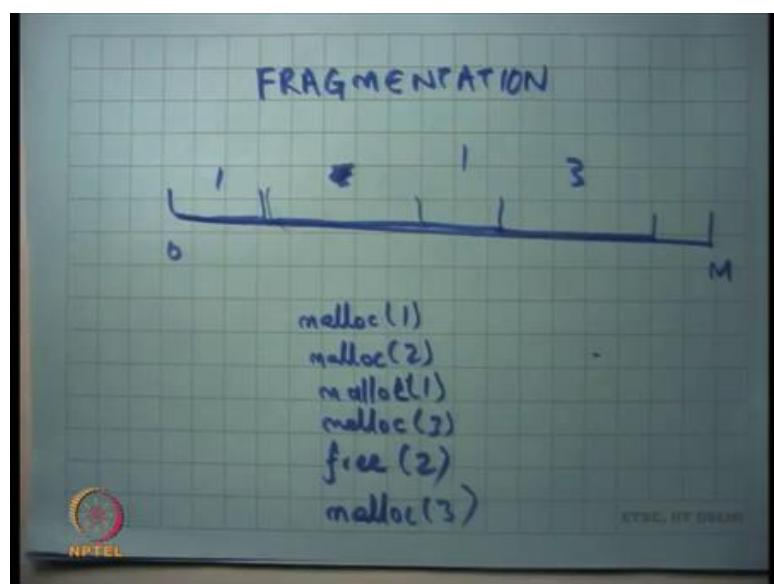
could not happen and this could happen if you know you have been mallocing too much and you actually run out of the space that the kernel allocated for it itself right.

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

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

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

(Refer Slide Time: 40:21)



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

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

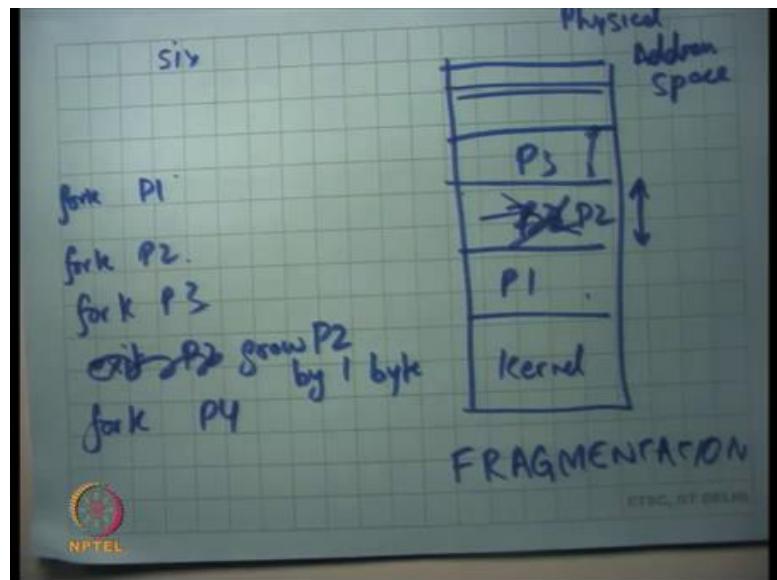
So, it is going to happen is this area is going to get freed and it is going to get added in the free list and then I say malloc 3 again and so, what happens is this is a region of length 2, you have asked for 3 bytes and I cannot just give you these right. So, what because there is no contiguous here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

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

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

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

(Refer Slide Time: 42:31)



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

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

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

Student: Shift.

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

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

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

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

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

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

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

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

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

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

Right.

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

Question.

Student: (Refer Time: 47:31).

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

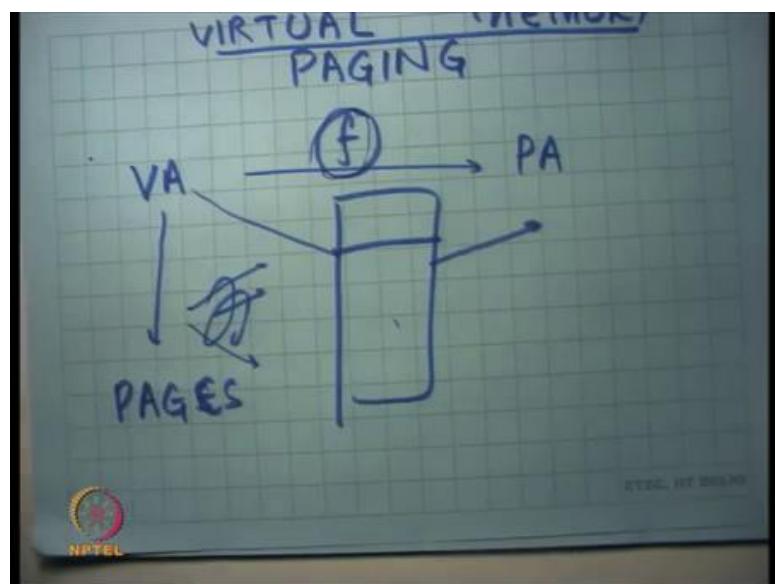
So, when I do fork, I have no idea what this process is going to actually execute. So, I just you know I just create an address space and you know in segmentation what I will do is I will probably create the address space is equal to the parents address space, but let us just exec an executable that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

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

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

Student: You answered that.

(Refer Slide Time: 48:45)



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

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

And, so what does what is the most general function? What is the most general function is for each byte basically have something which says you know this is where you should

this is where this byte lives this is where this byte lives, but the at the data structure if you to store this mapping will be bigger than address space it itself right.

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

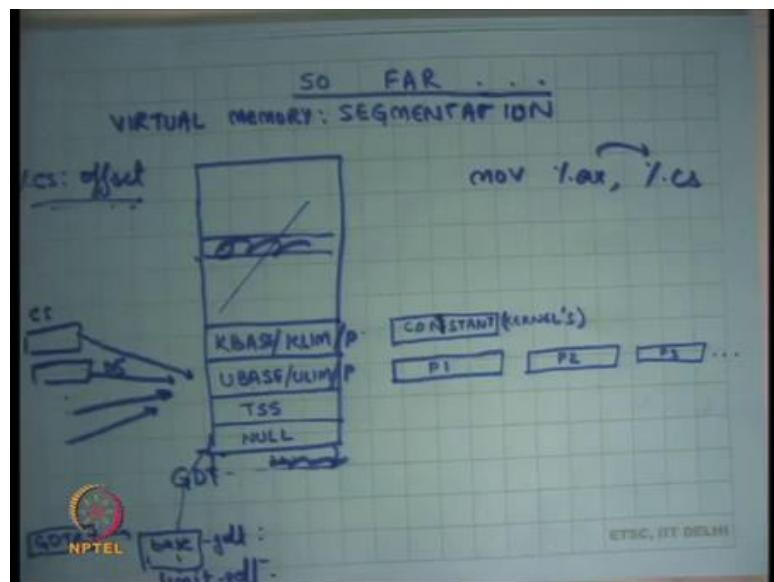
So, let us stop.

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

Lecture - 10
Segmentation Review,
Introduction to Paging

Welcome to Operating Systems lecture 10.

(Refer Slide Time: 00:33)



So, far we were looking at segmentation as a way of implementing virtual memory right. This notion where there is a separation between a virtual address and a physical address and there is some translation logic that goes to convert a virtual address with physical address. This system is called a virtual memory system. And we saw segmentation as one way of implementing virtual memory right, where the translation was rather simple, we just added base to virtual address to get a physical address.

And, we said the way it works is that there is a table which is stored in memory which is called the global descriptor table which contains these descriptors right. And this table can be as large as 2^{13} entries, where you know each of these is one entry. Each of these entries contains a base address which will be a physical address and a limit which will say you know what is the size of this particular segment?

So, each descriptor defines a segment and the program itself is going to you know, is going to dereference one of these descriptors using the segment selectors which are these segment, which are stored in these segment registers that are cs ds es fs etcetera.

And, you know they are going to dereference these descriptors here. And we also said one typical way of organizing the segment table is the following. You know you have some extra entries in this to store the task segment and null entry; we can ignore this for the time being. But there will be you know two descriptors, one will be for the kernel, the kernels base, and the kernels limit. And the other one will be for the user, users base and user limit right at any point of time.

So, at any instant of time you will have these two entries know in a typical operating system which is using segmentation to implement virtual memory. And, what happens is typically that the kernel base and kernel limit remain constant throughout the execution of the system right, because they refer to the kernels space. And the kernel always maps itself in a certain address space and in the physical memory and the kernel is always you know is always mapped, and it's always the same contents of the kernels that are always mapped.

But user base and user limit descriptor keep changing over time because you know context switches happened between one process to another process. So, you know for at one instant it could be the P1's UBASE and ULIMIT which are loaded in this descriptor. After a context switch it will P2's UBASE and ULIMIT which are getting stored in this over it overwritten in this particular descriptor right.

So, the kernel remains constant and the users the processes descriptor or the processes the address space which is for the user is defined by the by which process is currently running. And that keeps, that gets swapped on every context switch and that happens by the kernel. Both of these needs to be mapped at any point of time because you know as we have seen control can transfer from user to kernel if some event occurs like an exception or an interrupt or a software interrupt and in which case immediately you need to dereference the kernel side of that space.

For example, the CS in that in the IDT table could be a kernel could be must be pointing to the kernel descriptor and. So, both of these are mapped the way you prevent the user from accessing the kernel address space is basically by having another field here, which

is the privileged level they call it P. So, one of them will say you know I can only be accessed in privileged level 0, and the other one can say I can be accessed in either privileged level 0 or 3. So, you know it does not. So, that way this you know I when if the processor executing in unprivileged model only be able to access the dereference through this descriptor and not others, alright.

Also, typically now I said that there are 6 segment registers CS, DS and so on right and each of them could actually be pointing to a different descriptor here, potentially right. CS could say I want to point to descriptor number 2, DS could say I want to point to descriptor number 3 and so on right, assuming you know privileges are respected etcetera.

And, so that can potentially allow one process to have six different segments in physical memory and so that can solve some of the fragmentation problem that we discussed last time. But this kind of an organization where you divide your address space into multiple segments actually complicates the programming model, because now the compiler needs to worry about exactly how does this address live in this segment or that segment etcetera and it needs to generate code appropriately right.

So, typically what is done is that all of these actually mapped to the same segment to have this, to give the abstraction of a flat address space right. So, the compiler can assume a flat address space and all of these actually mapped to the same sort of descriptor and so you the process just sort of sees one flat address space from 0 to max right. So, it simplifies the programming model, but it has the problem of fragmentation as we discussed last time. Now, the entire process has to live in one contiguous segment ok.

Also there was a question last time about you know the GDT being too large you know, 2^{13} entries is too large and I am only going to use 5 entries out of these two 2^{13} entries. So, actually you know I emitted a piece of detail, the GDTR actually does not point to the base directly, it points to another memory location which contains the GDT base, let us call base GDT and limit GDT.

So, base GDT points to the base of GDT and limit says for the size of GDT. So, that way you know you do not need to waste all this space you can say that you know my GDT is only 5 entries or 10 entries and so. So, the hardware actually gives you a way of

saying that specifying the size of the GDT, and the maximum value of that size can be 2^{13} ok. So, you are not really wasting space in that sense alright. So, question.

Student: Will there be any point in time when we will be using all the space of GDT?

Will there be any time? Can it happen that I use all the space in my GDT? Depends on your kernel model, in the model that I have I have described so far, it is not possible right? With a number of descriptors that can have is actually constant.

Right is just these 4 registers descriptors for example. So, it is not possible in this model in another model it may be possible in which case the kernel should have logic to handle it ok, but let us just let be this model and say no it is not possible another question.

Student: Is the KBASE and KLIMIT constant for all the entries?

Are the KBASE and KLIMIT constant for all the processes is that your question?

Student: Yes.

Yes.

Student: The same entry.

Yeah. So, KBASE and KLIMIT do not change on a context switch ok. So, on a context switch the kernel remains as it is, it is the processes address space that you are changing.

Student: Sir, one more question.

Yeah, one more question.

Student: Base GDT and limit GDT can likely find to a physical address when we do not need a base GDT.

Yeah, so, these are these are actually physical addresses alright. So, these are you know let us call them physical addresses for now. So, and base GDT saying here is where the GDT starts, and limit GDT saying this is where the GDT ends, that is all.

Student: We can just have limit GDT we do not need base GDT.

How do you specify where it starts?

Student: Because we have GDT here which points to the base (Refer Time: 08:31).

No. So, that I am saying GDT I will point to this structure and this will contain the base and limit.

Student: We can have limit in the table itself right, GDT?

You are saying you know why I need this kind of thing; I could have you know GDT I could GDT could itself have some notion here which says limit. It is a matter of you know, it is a matter of fact that this is how it is done on x 86, but you could design something else ok.

Student: So exactly what is the (Refer Time: 08:59).

So, it basically says that you know everything above this is not possible. So, for example, if I did not have a limit GDT and let us say the I have assumed that the limit GDT will be 2 to the power 13 always. Then what will happen is the kernel will have to reserve that extra space and you know invalidate all these entries. So, you know there is a certain bait in these descriptors which says this descript is invalid. So, you will have to say an invalid and all these things.

If he does not do that then that space can be used for some other data structure and if data structure you know so a process can now manipulate those things right. Because, let us say there was some other data structure living in this space and now the process actually pointed to point here then you know he can now access something he is not supposed to access for example.

So, if there was no limit then the kernel would have had to allocate the entire 2 to power 13 and just two sorts of not waste that space, you the hardware allows you to specify limit. So, so how are UBASE and ULIMIT accessed? They are basically accessed using a virtual address right. The virtual address in this case is basically of the form of a segment register colon some offset, and offset could be specified in one of the mode immediate you know indirect or displaced whatever.

In either case CS actually goes here. So, what happens is offset gets added to base and offset gets compared to limit right. And, that is how the actually dereferencing, accessing the GDT entries on each a memory access ok. The user is not allowed to read these this GDT directly, its only allowed it only gets dereferenced through this mechanism of virtual memory translation or MMU ok. The user cannot access these entries directly no, in case of context switch does the limit GDT. You mean does UBASE and ULIMIT change?

Student: No limit GDT change.

Oh. You mean this? No this does not change. So, these remain constant I mean. So, it depends different, different operating systems can implement it in a different way, let us say the way we are were talking about it is that the GDT remains constant. It is just this entry that is changing, everything else remains constant right it is just just particular descriptor that is changing on a context switch question.

Student: Sir if a GDT stays constant then why did we need to store it anywhere? Like if we know the base GDT and we know the size of GDT that is required for the entire operation, why do we need to store it under a base limit?

Why do we need to store limit GDT to because we need to tell the hardware what the size of a GDT is right? because the hardware that is going to actually dereference. So, how is you, how do you go a do this? Is going to say, let us look at the selecting CS and added to base right and he is going to compare it against limit. So, if there was no limit then you know CS would point here. And so now, it is it is the responsibility the kernel to ensure that you know all these entries are invalid.

Student: No, but sir, like kernel will always have just 6, 6 segments selectives, user will UBASE and ULIMIT will always have 6 segments. So, basically the size of the GDT will always stay constant right?

Let us say in our model the size of the GDT always stays constant. So, yes absolutely.

Student: So, in that case like we do not need to really store limit GDT anywhere we can just like build it built it into a model itself (Refer Time: 12:18)?

No. So, this limit GDT is a way for the kernel to tell the hardware, that is all; I mean the kernel is not storing it for its own purpose. The kernel already know that it is 6, its telling the hardware that its six no it has to tell the hardware in some way and that this is a way of telling the hardware look you know there is only six entries that you should be accessing ok.

Student: Sir, in this model (Refer Time: 12:43) when your CS remains always the same? CS.

CS DS yes everything remains the same. In fact, you know in this model when I context switch between two processes the segments selectors remain the same. They always pointing to let us say the third entry in the stable right. It is just the entry that is changing every time.

But you know yeah it is possible that you have you know it is possible that the kernel allows you multiple UBASE and ULIMITS in which case the program is allowed to change between multiple things; great. The next question the next thing I wanted to point out was that on every memory access of this type right, let us say CS or DS offset whatever what is required is CS is stored on chip. So, you know there will be some logic that dereferences CS to get a selector, but the GDT itself now that that CS the selector is going to get added to the base of you know of GDT.

So, but to get to the base you need to dereference GDTR and the base is actually, stored on memory. So, you actually go to memory to get the base right, when you add it to CS then you get the descriptor, then you get the base from the descriptor once again you make memory access to get the base from the descriptor because the descriptors stored in memory.

And now you add the offset to the base and compare the offset to the limit and now you do a final memory access which is base plus offset. So, you know one memory access had how many more memory accesses? Two extra memory access right one for dereferencing the base and the other for dereferencing the base of user one for dereferencing the base for the GDT and one for base of the descriptor, right. So, that that sounds very wasteful right. So, what is the typical thing that is done?

Student: Caching.

Caching right: so, you have already run your computer architecture course you have seen caching. So, what is done is when you execute the load GDT instruction these values get cached on chip. So, there is some space on the chip which basically allows you to cache these values on chip ok. And plus you know the semantics of this are that if you change the kernel you know let us say changes the base limit to something else, then it does not immediately get you know updated in the chip because there will be a lot of logic that is required to do that.

Instead what happens is you know GDT the cache can remain out of sync from the memory. So, if you want to actually recache it then you should execute that LGDT instruction again at the new base and limit right. So, its caching, but it is not you know they do not remain consistent. So, the cached value and the real value in memory do not necessarily remain consistent you can reload the cash by re executing the low GDT instruction ok.

So, what does this kind of caching? Is it write back, write through?

Student: We have not done this (Refer Time: 15:38).

You have not done write back or write through questions?

Student: (Refer Time: 15:43) write back.

This is right back is this write through.

Student: (Refer Time: 15:52).

No, it is not written through right because I can actually go and write to the memory without the cache knowing anything right, write through basically means I have all my writes have to go through the cache. So, I go through the cache and I first update the cache and then all the things that I update in the cash go down. So, actually I can have direct access to the main memory. And so, cache is actually can be out of sinks. So, it is not write through.

Is it write back? It is not right back either because, once again I actually changed the value underneath and then update by using the LGDT instruction right. So, it is not, it is neither write back nor write through its you know it is another kind of caching you know

basically there you explicitly invalidate, or explicitly load new values will be cash right. The cache is actually just acting as a read only cache for you know.

So, that the cache is accessed in only read only mode, the only way you write it through explicit invalidation of the old value by re executing the LGDT instruction or you know invalidate instruction. Its neither right back nor right true, but you know it is a different type of cache alright.

Similarly, when you load values into the registers CS DS etcetera these the descriptor values base and limit get cashed on chip right just like this the descriptor values get cached on chip. So, you do not actually make memory access to each time you make a you do not actually dereference memory to get these values each time you make a memory access you have it on the processor cache.

Once again, the semantics are that when I load it gets loaded into the cache on locally on the processor, but if you change it later on that cache still remains retains the old value alright. If you want to override that old value, you need to re execute you know you need to reload the segment register using some instruction like move let us say ax to cs right. So, when you reload the segment register the cash gets freshly loaded, but if you change the memory underneath the cash remains keeps the old value ok.

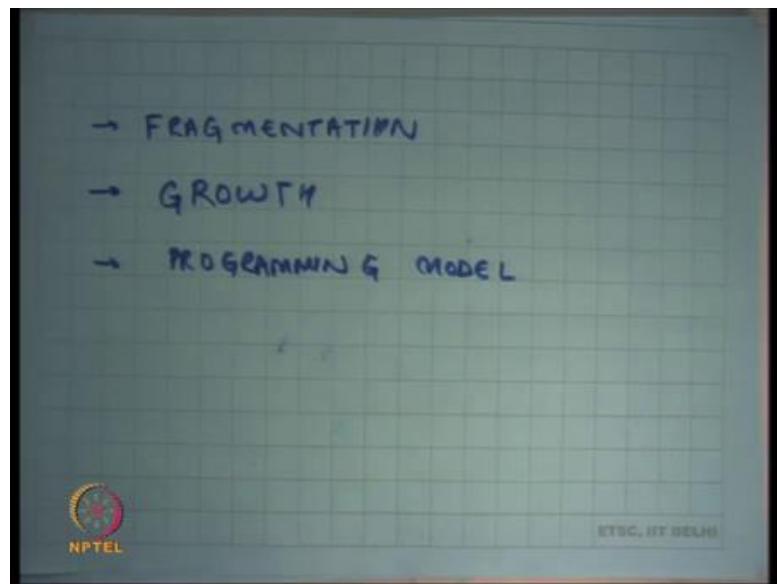
So, this is the caching model. So, with this caching model you know every memory access will actually make only at most one memory access usually typically right. Because you know each time you execute this address the GDT has base stored unlimited stored locally you can add it to the selector to get the descriptor value the descriptor value itself actually is stored in the locally. And so you can just get the UBASE from there or KBASE depending on whether it is a kernel segment or a user segment and you can and you actually just go to the memory for the real physical address and nothing else right.

So, this way you know you also have to worry about performance and so you care about performance, but this caching model also means that the OS needs to be careful that the cashed value and the memory value do not go out of sync right. So, GDT remain in the in this kind of model that we are talking about GDT remains constant. So, it only gets initialized once it gets cached once was does not need to bother really.

But and in fact, even KBASE and other descriptors are actually only remain pretty much constant it is only this user base and user limit that are actually changing and so the on every context switch what the OS will need to do is reload these segment registers again alright with that particular descriptor value. So, that they actually now get the new cash values before it actually context switch is back to the process ok.

So, that was segmentation and segmentation allow you. So, nice thing about segmentation or virtual memory in general is that it allows separation between physical address space and virtual address space. And our virtual address space can be this uniform address space starting from 0 to some value and the physical and you can place the processes anywhere you like at runtime and the same program is going to run with the assuming that it is actually running on a uniform address. But it has problems, the first problems we saw was fragmentation right.

(Refer Slide Time: 20:05)



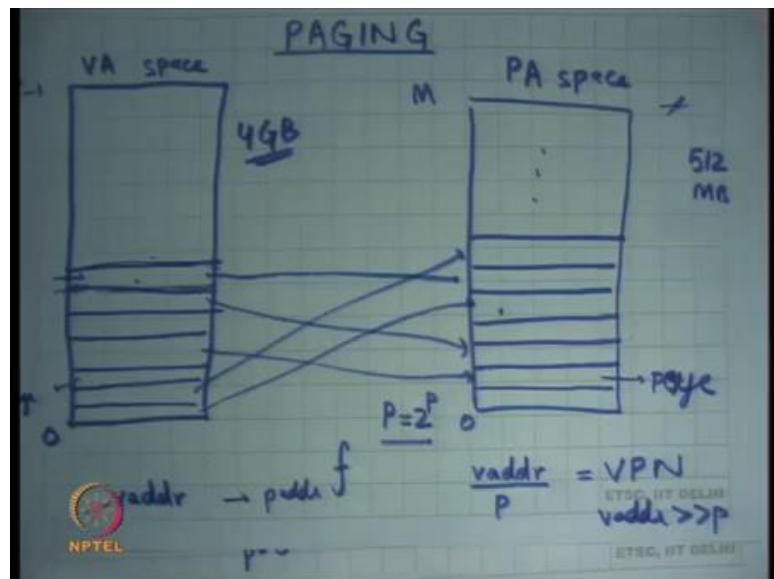
So, because a processes address space needs to be contiguous if different processes of different sizes are created, then very soon my physical memory can get fragmented and I could be wasting space. So, new processes may not be able to get admitted because you know the spaces that I have the holes that I have are smaller than the size of the process, even though the total space I have could have accommodated that process.

So, that is problems of fragmentation, it has problems with you know things like process growth. If I want to grow a process it is very difficult because I have already placed the

process somewhere and now the other processes which are you know around this process. So, now, I want to grow the process the only option that I have is either move other processes up or down or move this process somewhere else right and in both cases that requires physical copying of the entire memory of the process from one region to another in physical memory that is a lot of work.

And thirdly you know some of these problems can be alleviated by having multiple segments, but multiple segments have a problem that the programming model becomes complicate right. The programming model if you have multiple segments then the compiler has to worry about you know which segment I have to access as opposed to just saying here is an offset and that is it and I am doing that ok.

(Refer Slide Time: 21:41)



So, paging is another way of implementing virtual memory alright. Here what is done is let us say this is the physical address space, I am going to call it PA Physical Address space pa space alright. What it does is, it divides physical address space into fixed size chunks right, called pages. So, each of these sorts of frame is called a page right and then it divides the virtual address space also into pages, are the same size ok. So, this also page and then it has some function in the middle f which is implemented through a table, a lookup table which maps a page in virtual physical address to a page in virtual address a page in virtual address to a page in physical address and so on.

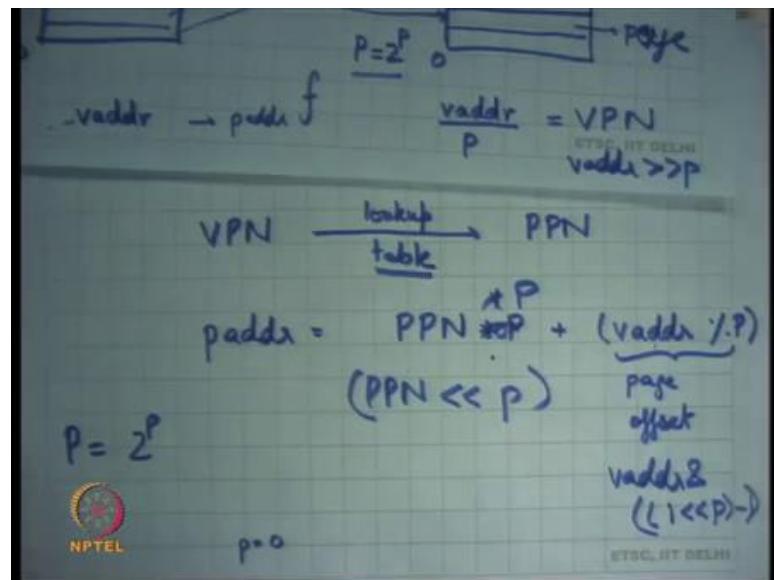
So, completely arbitrary mapping alright; so, the virtual address space also you know just like before start from 0 to some value, let us say you know let us say it goes all the way 2^{32} minus 1 because that is the maximum addressing that you can do anyways right because it is a 32 bit machine. So, the maximum size of the operand that you can have for a memory address is 32 bits because the registers are 32 bits, the immediate operand is 32 bits even you execute in displace mode it gets wrapped around to 32 bits alright.

So, so the virtual address space goes from 0 to 2^{32} minus 1 and the physical address space goes from 0 to whatever maximum you have alright depending on how much memory your machine has whether has 500 and 12 MB 1 GB. So, that is the value of M right. And this function is going to map these pages in this address space to pages in the physical address space ok.

Now, the program just works as before program just assumes that it has this virtual address space, it just know once again it just says I want to access address vaddr I want to access a address called vaddr, what happens is you go you say oh vaddr is here let us say. So, you say it belongs to page number this right, you basically do some calculation to figure out is this is the page number that it belongs to and you figure out on this page number is living in this page number of physical memory. So, you do that translation and then you can convert that vaddr into a paddr ok.

So, what is the nature of this translation? Well let us say the page size was p right and let us say the page size was p and p is you know equal to 2^p . So, let us say p was some multiple of or some power of 2 just to make things simple, because you know are, we are working on a binary machine where bits are binary. And so, so if I want to convert vaddr to paddr what I am going to do is I am going to say vaddr divided by capital P that is going to give me my page number in virtual space. So, I am going to call this the virtual page number VPN right.

(Refer Slide Time: 25:41)



And I am going to convert VPN; I am going to look up a table to convert VPN to PPN ok. That is a physical page number and now I am going to say now paddr is equal to PPN into P plus vaddr mod P right. That gives you a one to one mapping between assuming there is a one to one mapping between in VPN and PPN this gives you a one to one mapping between vaddr and paddr right.

So, basically you divide vaddr into a virtual page number and an offset inside that virtual page right. You convert VPN to PPN and then you add that offset. So, this is the offset this called the page offset right. So, you divided vaddr into a virtual page number and a page offset, then you converted VPN to PPN using a lookup table and then you convert computed paddr as this right you just dereference memory on that page and added the page offset to that ok.

If P is the power of 2, then these operations are very simple, vaddr by P is nothing, but vaddr shift left by p bits small p bits right and we had a $\%P$ is just vaddr and you know let us say one lesser than P minus 1 right. So, this gives you, its it is a mask which has the last small p bits set to one and everything else is set to 0 alright. And so, and PPN into P is nothing, but PPN shifted left by p bits alright.

So, now, the question is how big should this P be, or you know consequently p be? So, how big should it be if its 0, p is 0 which means P is 1, then you know I have extreme flexibility in where I can place each and every byte right, I can place each byte

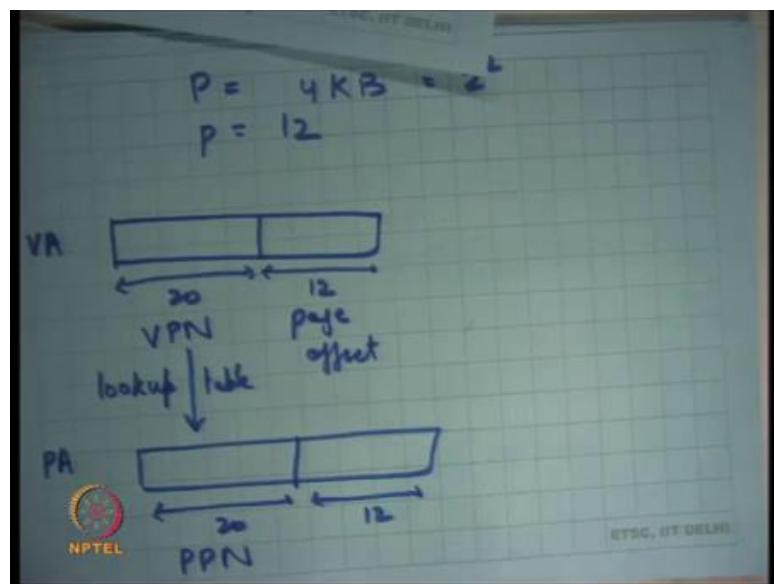
completely anywhere in the thing. But the look up lookup table how big is the lookup table going to be its pretty big right because for each byte I need an entry which maps it.

So, on the other hand if I have p as a very large number let us say P was P was equal to let us say let us say P was equal to 10 megabytes or. Power of let us say 32 megabytes ok. So, let us say P was 32 megabytes then you know firstly, I can now my lookup table becomes really small right because what is the size of this address space its 0 to 2 to the power 32 minus one which is four gigabytes right and the size of memory is also limited by whatever physical memory I have.

So, let us say if I had only 512 MB of physical memory then the number of pages I will have if I have a 32-megabyte page is 512 by 32 which is let us say 16 right. So, 16 pages in the entire thing; so, my lookup table is going to be just mapping 16 integers here to 16 integers there right. So, my lookup table is only going to need 16 entries if my pages that big.

But, the problem with that is if most of my processes are really small, then each time I allocate a page process I need to actually allocate at least 32 megabyte of state space number 1. So, I am wasting a lot of space in the middle right and ok. So, you know here is an engineering decision to be made, what is the size of a page and the typical size of a page that is used in modern hardware is 4 kilobytes alright. So, capital P is equal to 4 kilobytes and small p is equal 12.

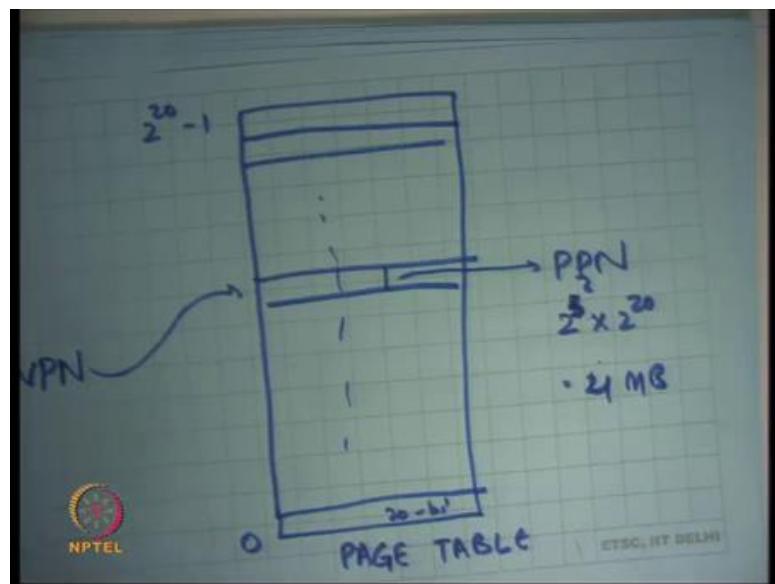
(Refer Slide Time: 30:25)



So, just 2 to power 12; so, we are going to you know as we go along the course were going to see why you know a number like this makes sense you know it actually depends on the physical characteristics of the hardware etcetera, but let us just first understand assuming there are pages 4 kilobytes, how does this mechanism work? So, pages 4 kilobytes what happens is a virtual address is 32 bits of which the last 12 bits are called the page offset and the top 20 bits it is called the VPN right.

And, similarly in the physical address this is the PPN. So, now, I need to design a lookup table which maps VPN to PPN. So, in other words I need a mapping from 2 to the power 20 numbers to another 2 to the power 20 numbers in the most general case. So, one way to do that is to have a table of 2 to the power of 20 entries right. So, I could have one big table.

(Refer Slide Time: 32:01)



Now, let us call this the page table, which has 0 to 2 to the power 20 minus 1 entries ok. And I am going to take the VPN and use it to index this table right and what about value whatever value is stored there that is going to be called my PPN ok. That is one way to do this. How big is this table?

Student: (Refer Time: 32:42).

So, each entry will need to be 32 bits right. So, that is two to the power 5 into the number of entries, yeah. So, into 2 to the power 20 that is equal to 2 to the power 23 or 8 MB.

Student: What each entry of the power 40 bits, 20 bits for the, for the source and 20 bits for the rest?

The question is should not each entry have 40 bits instead of 20 bits.

Student: (Refer Time: 33:13).

No, because the VPN is actually being used to index the thing right. And so, you save space in that sense by doing this kind of thing. The another question could have been you know why do I use this thing where I have a huge page table, where I am actually assuming the VPN can be used to index set why not have something more efficient like a link list which has all these VPN to PPN mappings right. That is likely to be smaller in the common space, but you know it is a time to actually walk through that linked list that is going to become the bottleneck.

So, as you can imagine you are going to actually do this translation on each and every memory access and. So, the most important engineering decision you to take is basically to minimize this translation overhead right. You want to minimize the translation overhead; it does not matter space can be optimized in other ways if possible ok.

Student: Why is the PPN 32 bits?

Why is the PPN 32 bits, it could have been 20 bits right, but you know 20 bits the memory itself is byte addressable right? So, if I say its 20 bits here, then the address of the first entry becomes 20th bit and does not makes I paddr access a bit in memory I can only access memory at granularity of a byte number 1 ok.

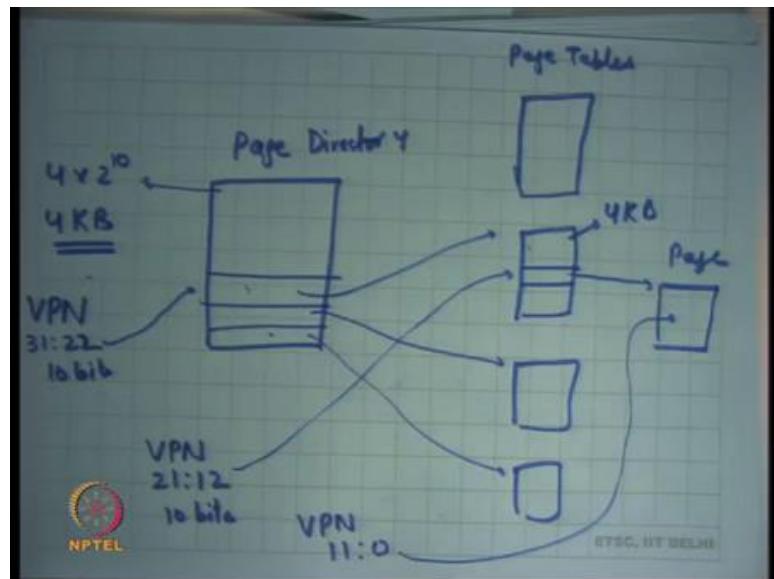
You could say why does it need to be 32 bits why cannot it be 24 bits for example, right which is 3 bytes, well that is possible, but actually that that also complicates hardware you know it is better to actually have power of 2's. And the other thing is you actually use that extra 12 bytes for other purposes extra 12 bits for other purposes which were going to discuss later ok.

. So, that is a lot space. So, that like let us say this is truth of 32, 32 MB right, I am getting it right ok. So, 32 hold on hold on. So, there is 2 to the power 2 right; so, 4 MB right that is 32 bits which is 4 bytes so 4 MB off space. So, clearly 4 MB off space cannot be allocated on chip right. We said chips can only have some registers which are

few 10's to 10's of bytes. So, 4 MB of space paddr be allocated in chip. So, this data structure has to live on memory right.

And, but if the state data structure has to live in memory you are saying that every process will have its own page table because every process will have its own virtual address physical address mapping. So, every process will have need at least 4 megabytes of space, to at least store the page table typical process sizes will be few 10 of KB to 1 MB to you know things like, processes like grep and ls all these are very small processes and you know if for every process I am actually creating a page table that is 4 megabytes of size that is a lot of wastage.

(Refer Slide Time: 36:13)



So, what is done is basically you divide the page table into a 2-level page table. The first level is called the page directory and the second level are called page tables, where each of these entries in the page directories points to a point to a page table and so on.

So, once again what I do is I look at the VPN and I take the top 10 bits of VPN. So, let us say VPN bits 31:21 or let us say 22 to 31st bit and we use that to index the page directory. I get some address this contains the physical address of the page table and then I look and then I use VPN, 21:12 to access the corresponding page. And so, I get a page and then I use VPN 11:0 to offset into the page right.

So, just I mean what I have done is I have divided the page table into a 2 level hierarchy, I said why not why not just have a list or a tree to store this and you said you know having a list or a tree is actually going to increase complexity of hardware. But at the same time, you need to strike a tradeoff between how complex the hardware needs to be and how much space you can waste.

And so you know the designers thought let us just let us just divide the table this large table of 4 MB into a tree of tables, but allow the tree to be the tree must be of size 2, or depth 2. So, use it; so in this case I was using 20 bits to index into this page table in this case I am using the first ten bits to index into the first page directory and I am using in the next 10 bits to index into the second page directory or page table.

So, this way you know if I have a process which is actually taking only you know let us 100 KB of space. So, it only needs 0 to 100 KB mapped in the address space. So, will happen is you know 1 or 2 of entry the entries here are going to be filled and all the other entries are going to say they are not mapped, right and that way you save space.

So, only one or two entries are filled in the page directory and you only have one or two page tables in the in that case and the total amount of space that you using for this page table structure is let us say 3. 3 of this right, 1 here and one of each of these let us say two of these one or one or two of these depending on the size of the process. Let us see what the size of the page directory itself I am using 10 bits to index into the page directory. So, how many entries does it have?

Student: 2 to the power 10.

2 to the power 10 right, and how big is each entry.

Student: (Refer Time: 39:36).

Each entry contains a physical address of the page table. And how bigger physical address be? 32 bits. So, it has, but a physical address can be 32 bits, but you have also segmented the physical address space in to pages right. And so, the number of pages you can have in the physical address space is 2 by 20. So, really each entry needs to be 20 bits right each entry needs to be 20 bits. And so, but once again 20 bits is you know I is a is not a very nice number. So, we extended to 32 bits and so it becomes 32 bits 4 bites.

Student: 4 kb.

Right, so it becomes 4 into 2 to the power 10 which is 4 kb.

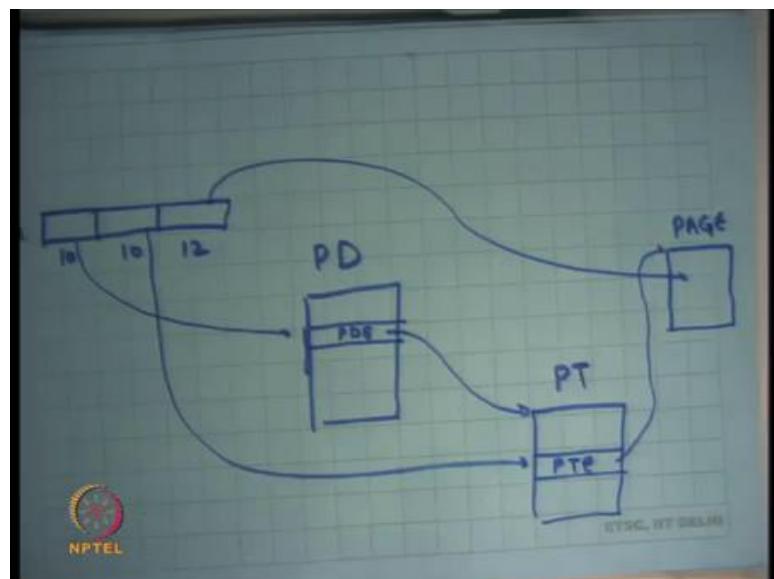
Student: (Refer Time: 40:23).

So, that is the size of page directory and interestingly that is the same size of the page right. So, the page directory itself lives on a single page. Similarly, what is the size of the page table?

Student: (Refer Time: 40:37).

Same thing right; you are using 10 bits to index in the 4-page table and each entry is 4 bytes. So, its 4 KB again, once again the page table itself lives on a single page right. And then you use that to index the page and you get the page alright. So, let me just review this, this is the virtual address VA.

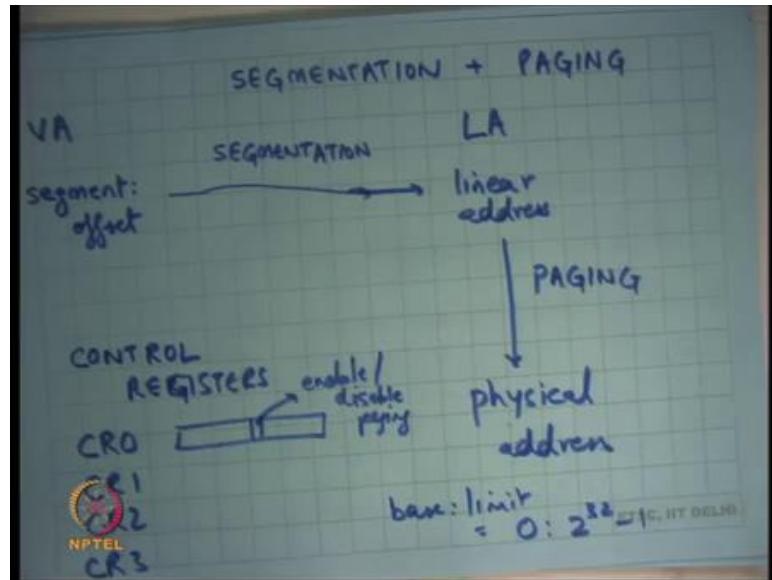
(Refer Slide Time: 40:59)



We divided into 10, 10 and 12 was 32 bit address, you use the first 10 bits to index into a page directory that I am going to call it PD Page Directory and you are going to get up PDE Page Directory Entry using this. You are going to get the contents of the PDE to dereference a page table. Let us call it a PT, and you are going to get the next ten bits from here to index into the table I want to get a page table entry right.

And now you are going to get a page from here finally, you got a page and you are going to use in last 12 bits to index into this page to get your final physical address. Let us see how this works on x86.

(Refer Slide Time: 42:13)



So, x86 actually uses both segmentation and paging alright. So, virtual address of the form segment colon offset, this is called a virtual address VA. Go through the segmentation hardware as we discussed earlier to give you another address which we have been calling the physical address so far were actually gives you what is called a linear address alright.

So, let us call it the linear address. The reason it is calling the linear addresses because now this address has no segment colon offset, it is just 1 number with 132-bit number right. So, it is a linear address space and now this goes through paging hardware to give you a physical address ok.

. So, it gives you both an operating system is free to use both; an operating system is free to use only segmentation in which case all it needs to do is disable paging alright. So, how do you disable paging? It has certain registers which are called controller registers CR0, control registers just a 0 CR1, CR3 let us say CR2 maybe more. And, you know there is a bit in CR0 which says enable or disable paging. If you disable paging only using the segmentation hardware and that is likely to be quite fast right, because you are

not going through the paging thing which is which is involves looking through the page tables.

But actually, as you can discuss later paging is also not all that slow and you know there are ways to make that fast and this is basically what is typical us it has advantages. So, an operating system is free to use only segmentation or an operating system by disabling this bit. All in operating system is free to use only paging right, in which case what does it need to do? You know all it needs to do is basically set all its you know set its segment selector that we discussed in our first slide ok.

So, all it needs to do is set its set all its base and limit to 0 to 2 to the power 32 minus 1 right, if all its segment descriptors it sets its base and limits to 0 to 2 the power 32 minus 1 its effectively disabled segmentation.

.So, in the in the next few lectures I am going to assume that the base and limit have been set to 0 and 2 to the power 32 minus 1 which means the virtual address is equal to the linear address. And the most operating systems actually do this right. So, most operating systems like Linux, Windows and the operating system that you are going to do in this class which are x86 (Refer Time: 45:33) and pintos will all set base and limit to these values. And so only use paging for implementing virtual memory ok.

Student: (Refer Time: 45:6).

What is the use of having segmentation? Well the hardware designer has given you a choice basically right, and the hardware designer designed the hardware before the operating systems were written on it. So, there was basically tried to say that I do not know what the operating system may want in future.

And so, he designed it and because of backwards compatibility it remained in the hardware. In fact, on the 64-bit architectures that have you know come up recently segmentation is not present. So, you do not have segmentation in this form, it is a very thin form of segmentation that you need because most operating systems are not using segmentation.

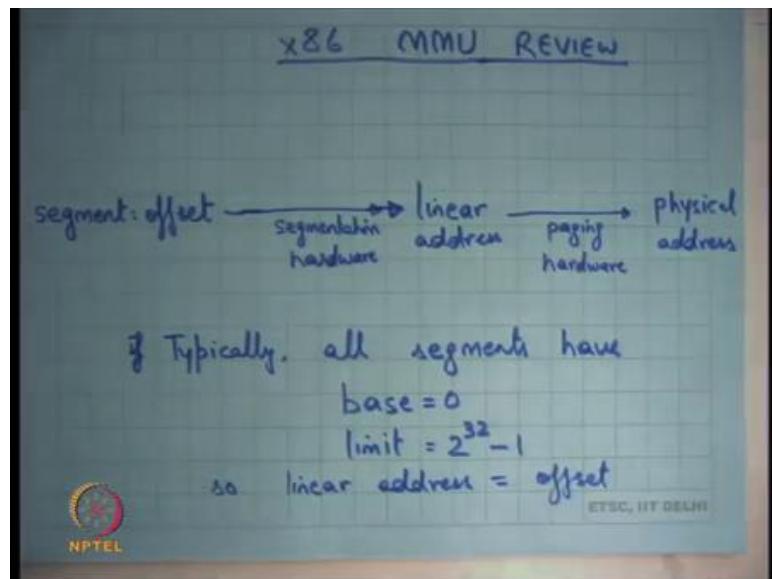
Alright; so, I will stop here and we going to discuss more about paging in the next lecture.

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

Lecture - 11
Paging

Welcome to Operating Systems Lecture 11. So, we have been looking at the Memory Management Unit in a typical hardware system and we have been looking at the x 86 MMU in particular.

(Refer Slide Time: 00:35)



And we said the x 86 MMU implements both segmentation and paging right. So, a virtual address is always of the form segment called colon offset. Now CS colon offset, DS colon offset etcetera. The segment will point to a segment descriptor. We have seen this in detail and then, the offset is going to be added to the base which is stored in that segment descriptor to get a linear address right.

So, far we had been calling this is a physical address, but it really is a linear address if paging hardware is also enabled. So, further this linear address goes through another level of translation which we call paging, to get you a physical address ok. Most operating systems or most systems in general, actually today do not use segmentation hardware for this kind of translation; I mean in the way that it in the sense that they all

they typically said base is equal to 0 and limit is equal to you know whatever the maximum value is in a on a 32 bit machine for all the segments.

So, you get a flat address space, irrespective of what segment you are going through, whether you are going through CS, SS, DS does not matter; it is the offset that counts right. So, basically effectively that means, that your linear address becomes your offset ok. You could imagine even operating system which is actually using seg only segmentation and not paging.

We have discussed this previously right; in the discussion that I had been having so far when we did not discuss paging, we said you know it is possible to implement virtual address spaces using segmentation and in which case you know linear address becomes equal to physical address. And, it's possible to do this, you just keep swapping the users descriptor and keep changing its base and limit depending on which process is getting loaded right.

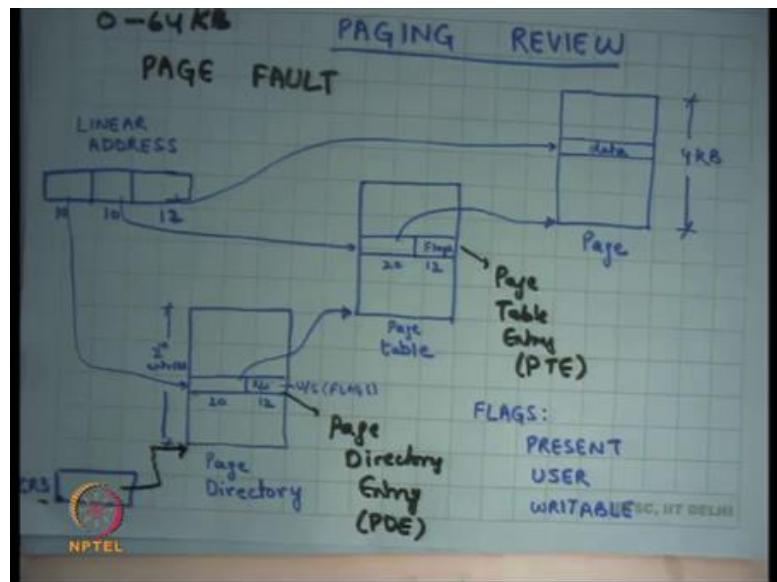
So, it is possible to implement a complete MMU using only segmentation. The only problem is it has some drawbacks and the drawbacks are for example, fragmentation becomes a problem, growth of a process is very tricky. And, plus you know if you are using multiple segments, then the programming model has to worry about which segment I am currently accessing and all that right.

On the other hand, if you have a flat segmented model which mean which all the segments are have exactly equal base and limits, in that case you are you know you the fragmentation problem becomes more severe alright.

Yet another operating system can just use paging hardware in which case he just sets you know all these things to thing. It still uses a segmentation hardware for things like you know for understanding what a privileged level I am executing on, yes recall we call that the last 2 bits of the CS register, basically indicate whether I am executing in privilege level 3 or privilege level 0.

And so, that still holds, but apart from that the offset pretty much has a one to one mapping to the linear address alright. So, base is equal to 0. So, in which case he is only using the paging hardware and yet another operating system could try to use both right, that is also a possibility alright. So now, let us look at paging in more detail.

(Refer Slide Time: 03:37)



So, we say look paging gets rid of some of the problems of segmentations. The main problem was with segmentation was that it required the entire process to be in one contiguous chunk in the physical memory. It is an entire process memory should be in one contiguous chunk in the physical memory and so; it was creating problems with fragmentation and growth.

So, as opposed to that if it was possible to have a mapping between virtual addresses to physical addresses such that the flexibility is more in which case you know a process could be sprinkled across that physical address space. But, in the virtual address space, it maybe contiguous and you could basically have mappings, arrows going from the virtual address space to the physical address space right.

And, we said you know it is possible to do that, you know one has to design it carefully if; so, basically at what granularity are you going to do this mapping. So, we said let us call the granularity at which you are going to do this mapping a page right. So, for every page; so, within a page contiguous byte in the virtual address space will be also contiguous in the physical address space, but across pages that need not be true.

Now, question is how big should a page be? He said you know page, page cannot be so small that the data structure to store this translation is so big, that it is actually bigger than the actual address space itself that is you know that is ridiculous. On the other hand,

it should not be so big that you know you are wasting a lot of space. So, we said you know let us say that 4 kilobytes are a reasonable value for a page and then we said ok.

Then, if 4 kilobytes are a reasonable value for a page, what is the maximum size of virtual address space can be? It can be 2 to the power 32 right. So, 2 to the power 32 divided by 4 KB comes to 2 to the power 20. So, you can have utmost 2 to the power 20 pages in your virtual address space. Similarly, your physical address space let us assume can be utmost 32-bit flight. Let us assume that the bus can only take a 32-bit address on the physical bus which goes to memory. So, your physical memory can also be up to 2 to the power 32 bytes let us assume.

Although, you know it may be smaller depending on actually how much ram you have in your system. In any case the hardware needs to be needs to provision for the maximum possible. So, let us 2 to the power 32 on both sides. So, what is the number of maximum number of pages on both sides? 2 to the power 20 and so, you need a mapping from a set of 2 to the power 20 numbers here to another set of 2 to the power 20 numbers here right.

And so, this mapping we said is you know if I was to do it in one contiguous table, it will require 2 to the power 20 entries. So, I can just have an array which has 2 to the power 20 entries and each entry basically stores where this page is mapped.

This array will actually be pretty big it will be 4 megabytes if I were to do this and so, assuming that each process has a separate address space, each process will need a separate mapping. And, so for each process I will need to have a separate data structure of this type; but this sounds very large you know 4 megabytes for every processes a is very large because typically processes are smaller.

So, let us divide this mapping into a 2 level hierarchy right and the first level we are going to call a Page Directory which is going to create a mapping from the first 10 bits to a page table. And then, the second page and the page table that you obtain you are going to use the next 10 bits to get the actual physical address. We have divided the 2 to the power 20 entries into sets of 2 to the power you know into is divided 20 bits into 2 into 10 bits 10 bit look up.

So, the advantage of this is let us say a process has only is only mapping you know 64 kilobytes of space right. So, 64 kilobytes of space can be captured in you know 64

divided by 4 that is 16 pages right. So, 16 pages can probably just you know 1-page table can have up to 1 0 2 4 entries. So, 1-page table can actually have 1 0 2 4 pages. So, in just you just need to allocate 1-page table and fill in those 16 entries here and just have one entry in the page directory that is pointing to that table at the corresponding offset right.

So, let us say my process was having an address space from 0 to 64 KB alright. So, what will happen is the 0th entry in the page directory will point to a page table and the 0th to 16th entries in the page table will point to pages, all the other entries will be invalid ok. So, that way a process which was only 64 kilobytes large needed a page table structure of 8 kilobytes right, 2 pages each of these is a 4-kilobyte structure. So, you needed 1-page directory and 1-page table.

So, you know you needed 8 kilobytes to store the mapping for a 64 kilobyte process which has a contiguous mapping from 0 to 64 byte kilobytes; so, significant improvement over the 4 megabytes that we had earlier right. It is a simple thing you just divide 1 linear array into a 2-level tree and you basically only have pointers where there exists a mapping and for others you just say its invalid ok.

So, of course, the downside of dividing 1 single table into 2 is what? Number of lookups right. So, number of lookups that you need to do in memory or number of dereferences that you need to do in memory has doubled. Earlier you just had to do 1 dereference to get the physical address, now you have to do 2 dereferences.

First you have to do 1 dereference here, then you have to do a dereference here before you actually get a get to the data right. So, here is how it works on x86, there is a special register called the controlled register 3; we said there are some special controlled register then x86. So, there is a special register called controlled register 3 which points to the base of the page directory.

And then, you know any linear address that is computed through the segmentation hardware, the first 10 bits are used to index into this page directory to get a page table. The next 10 entries; bits are used to index into the page table to get the page and the last 12 bits are indexed into the page to get the real data right. The physical address is actually this value plus whatever this offset is right that is that will be the physical address and that is where the data is going to live in physical memory.

Student: Sir, why?

Ok. Why does the pointer to the page table need to be 20 bits? Great question. Because in.

Student: Each page table is also a page.

Each page table is also a page you have basically divided your entire physical address space into pages right and you have mandated that page directories and page tables will also be constrained to start only at page boundaries right. A page table cannot be across 2 pages right. So, you basically just statically created a partition of you know you are basically said every 4 kilobytes you have drawn a line and you have said that a page table has to live in one of those slots it cannot actually you know straddle 2 slots ok.

So, because and so, the number of bits you need to actually understand what which slot you have is only 2 to the is only 20 bits. So, similarly you know the page directory needs to be really only 20 bits. The page table can be addressed using 20 bits and the page itself can be addressed using 20 bits ok.

We also said you know even although these pointers are 20 bits, it makes sense to actually allocate 32 bits for the entire entry. So, this entry is also called the page, this is called the page directory entry, and this called the page table entry alright.

So, let us use the words PDE and PTE for them alright. So, both PTE and PDEs are 32 bits long. So, the last 12 bits are can be used for other purposes and in particular, these last 12 bits are used to store flags right. For example, just like in segmentation you can where you could store you know whether this segment is allowed to be accessed in unprivileged mode or not. You can say whether this page is allowed to be accessed in unprivileged mode or not alright. So, that is the user flag.

Firstly, there is the present flag whether it says whether the page is actually present right. So, so in this example let us say a process has only 64 kilobytes of mapped space from 0 to 64 kilobytes and it is a linear address. Then, the only the first entry will have the present bit set; all the other 4 you know 1 0 2 3 entries in the page directory will have the present bit zeroed out alright. That is what it means that the entry is actually invalid.

Similarly, in the page table the first 16 entries will have the present bit set, all the other entries will have the present bit set to 0; so, that is a bit which says whether there is, whether this page table entry actually has means anything or not; whether the hardware should actually consider walkthrough dereference it at all or not alright.

Then, there is another flag which says user which basically says whether I am able to whether I should dereference this page table entry, only in supervisor mode or am I allowed to dereferences page table entry either in supervisor or in user mode right.

So, you can do it either in user or privileged mode and the third thing which it has is whether this page is writeable or not right. So, it gives you another sort of added capability. You can actually the operating system can actually map certain pages as read only right and we will see you know why this is a very useful thing to have.

So, if a page has a writeable with zeroed out, then if a program tries to write to an address that dereferences through that entry, it should generate an exception alright and this kind of an exception is called a page fault. So, instead of a segmentation fault, now you are talking about paging. So, the new name is called a page fault. The meanings are similar. Basically, you are trying to access an invalid address, or you are trying to address access a valid address in an invalid way right.

So, a page fault can occur if you are trying to access a non-user page in user mode or if you are trying to access a non-present page or if you are trying to write to a non-writeable page, you know either of these basically mean that you know you generate a the hardware will generate a page fault.

And you know the page fault handler will do the similar thing. For example, it may kill the process, or it may you know convert the page fault into a signal that it gives to the process all these just like before right.

Student: So, is this page table are completely managed by the hardware or does OS also has role to play in it?

Ok. So, yeah, great question; are these page tables managed by hardware or does the OS have a role to play in it. The OS sets up these page tables. So, the OS sets up the CR 3

pointer for example right. The OS sets up the contents of the page directory. The OS sets up the contents of the page tables.

Before it transfers control to the process, but for each memory access the OS is not coming into picture. It is a hardware that walks this page table or the hardware that reads these page tables right. So, this process of actually converting a linear address to a physical address, it is called page table walking or walking the page table.

So, the walking off the page table is actually done by the hardware. The operating system sets up the page table ok. So, the hardware walks the page table and if there was an error, for example you know you are trying to read an entry which is not present; then it is an error page fault right. So, yeah so, basically each of these flags. So, when I say there is a flag in the page directory entry, let us say the present flag in the page directory entry is not set, what does that mean?

Student: It means (Refer Time: 15:38).

It means there is actually no page.

Student: No mapping.

In this, there is no mapping here right. So, if the program actually tried to access a linear address that has a first 10 bits map pointing to this entry which has a present bit not set, then you will generate a page fault right there. It does not need to do a second level page look up at alright; similarly, user and writeable.

So, I mean you know so it is possible that this entry was present, but it says this entry is not writeable. On the other hand, the page table entry corresponding page entry table says writeable; you know one of the entries says not writeable, the second entry says writeable. So, the net effect is basically the and of these two which means not writeable.

Once you know conversely this could be writeable and this could not be this could be non-writable in which case overall it is non-writable. So, you know the OS can do it do this in a very fine grain level at the page granularity. You can do all these writeable, you can mark all these writeable and then, you know only pages which are not writeable in this particular entry that those are non-writable otherwise others are writeable and so on; same thing for user and supervisor right. So, you could have a you know the page

directory entry itself could say I am I am a user mode entry which means anybody can access me, but some of these entries are user and some of these entries are supervisors.

So, once again you take the and of these two. So, if both of them are supervisor, only or both of the only both of them only a both the user bits are set can the user access that page right. If only if for any of these entries of user bit is not set, then the hardware should not be should generate a page fault, if it executing in unprivileged mode alright. So, let us see how paging can be used. So, we saw how segmentation can be used to implement processes question.

Student: What is the difference between segmentation fault and page fault like if we are not using segmentation then?

What is the difference between a segmentation fault and a page fault, actually at the Unix abstraction level a page fault is converted into a segmentation signal sigsev right? So, both of them actually mapped to the same thing ok, but at the hardware level you know on x86 different exceptions are used to indicate different conditions. So, there is a separate exception number for a page fault and there is a separate exception number for segmentation violation.

Student: If we are not using segmentation (Refer Time: 18:10) you should have (Refer Time: 18:13).

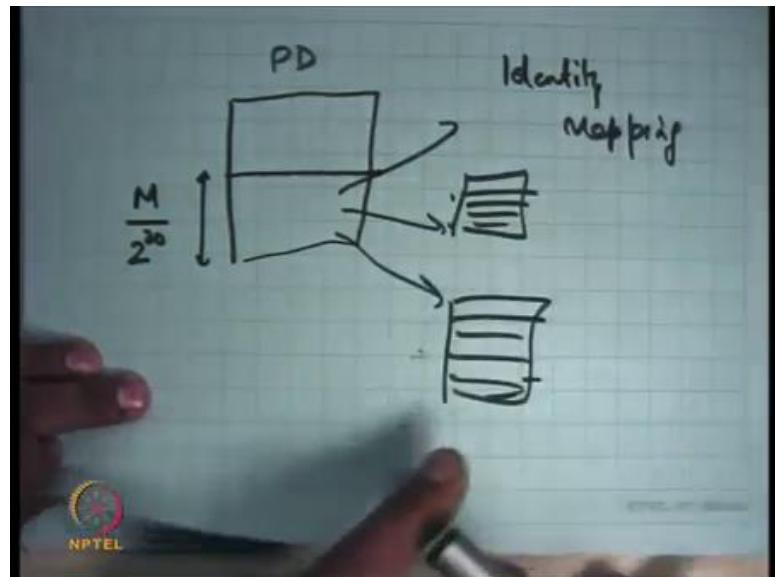
If we are not using segmentation, a segmentation violation should never occur. Well, that is not true because you know you are not using segmentation in the sense that you are not using it to segment memory, but you are still using the privilege bits or the you know the permission bits in the segment descriptor. So, you know if you try to execute for example, the user tries to load into the segment selector a privileged descriptor.

So, that will still cause a segmentation violation ok. So, it is still possible to have a segmentation violation even in this flag segmentation model, where you know you basically when the entire space into all the segments alright.

So, let us see how paging can be used. Well, one very simple thing is you set up the page table to basically have an identity mapping right. So, linear address x always maps to physical address x right and so, that is one way and how will you do that? You will just

say oh how big is my physical memory, let us say 0 through m. So, you know you will basically say 0 through M.

(Refer Slide Time: 19:19)



Let us say this is my page directory. I am going to set up you know the first M by 2 to the power 20 entries here right and then, each of these are going to point somewhere and each of them is going to set up all its entries, appropriately right to exactly the same address. So, you could set up a linear completely identity map between your linear address and your physical address right.

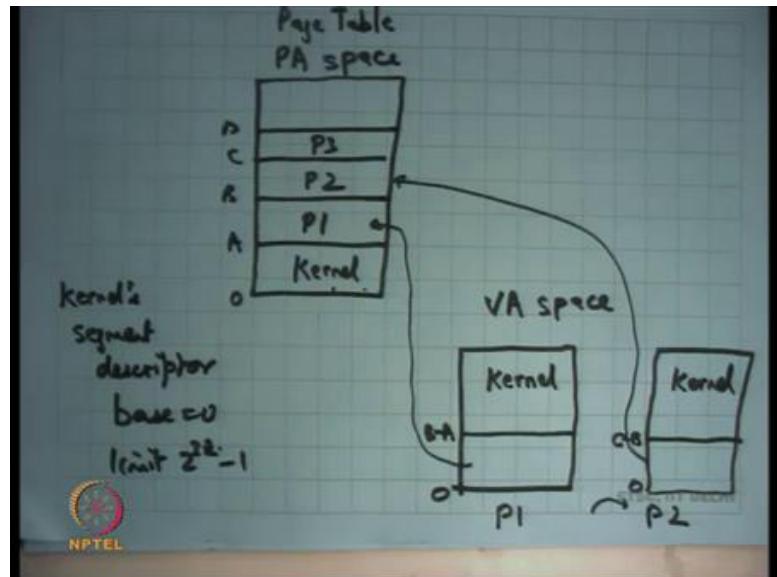
That is probably you know that is probably what you will do if you are not implementing multiprocessing and you are not having multiple processes and all that. For example, if you are you know using your system, your processor to implement some kind of an embedded system like for example, something which does not expect multiple processes to be running.

It is just doing when run the OS in one way and you have this paging hardware, you would probably just set up an identity mapping and or alternatively you could even disabled paging and so, it is the same effect right.

So, just to get a just to understand that you know you can how you are going to do this, it sorts of increases your understanding of how paging works. So, identity mapping is easy.

The other thing you can do with paging is you know set it up like segments right. So, you can say ok. So, here is my page table.

(Refer Slide Time: 20:53)



And let me just assume that you know it is a uniform sort of address space and so, what I am going to do is I am going to say let us say this part of the. So, let us say this is my physical address space; this is my PA space alright.

So, let us say this part of the PA is a or let us say this part of the PA is reserved for the kernel and this part of the PA is reserved for process P 1, this part of the space is reserved for P 2 P 3 and so on and let us say this is free ok. Just like what we had in segmentation and we were implementing and using base and limit. Let us what I am going to show you is basically that paging is more general than segmentation because you know you can easily do this and paging and so on right very clear.

What you are going to do is basically say that in your page table. So, let us say this is physical address 0 and a this is you know something let us say A B C and D right. So, in your page table and your page directory, you are going to say let us say in your in your let us say this is VA space.

In your VA space, so this was a PA space, and this is the virtual address space. In the virtual address space, each process should see a see the same set of addresses. So, what you are going to do is in the VA space, you are going if P 1 is loaded; then you are going

to map P 1 to this space right. You are going to set up the page table such that 0 to B minus A of P 1 maps to A to B in physical address space right. Easy to set that a you just set up the page directory entries and page table entries appropriately.

Then, let us say you switched to P 2, you will load a new page table alright and how will you load the new page table? You will change the CR 3 register right; you will change the CR 3 register. So, a new page table gets loaded and the new page table will have a similar thing, where it will say 0 to let us say C minus B and it is going to map this to this alright. So, you can just so in the segmentation model, each time there was a context switch we were overwriting the segment resist descriptor with a new base.

In this case, each time there is a context switch; I am overwriting CR 3. So, that I have I have a new page table loaded and I have set up the page table to basically emulate exactly what the segmentation hardware would have done.

Student: Sir, the changing CR 3 where changing page directory not page table right?

Yes, we are changing the page directory and also the page tables. So, we are changing the entire 2 level hierarchy right; so, assuming that there is no sharing between different page directories right, each page directory. So, you change the page directory and consequently you change all the pointers of the page directory also right ok. So, your when you context switch in the paging world, we are going to replace the CR 3. So, you are going to have a new page directory and you can you know set up your page directory or page table to do this mapping just like you were doing in segmentation.

The only difference is that in this case you know the page tables will be different right. In 1 case the page table mappings that entry is visually present in the page table, you know the or the pointers in the page tables will be different. There will be pointing to different regions in the physical memory basically question.

Student: Sir, sir if they are using two separate like for each process separate page directory and page table, there are 2 level hierarchy, we have to like in case we update one of them we have to make sure that those physical address or not mapped in (Refer Time: 24:46).

Yes. So, if you know each process has a page directory and a page and a page table structure and let us say I update one of them with a new pointer. I have the OS must ensure that this pointer wherever this page is pointing in physical memory that is not it does not that that page is not being pointed to by any other page table structure right.

So, that is that is possible for OS to do. No problem. The OS just keep some there is some bookkeeping, at these pages are belonging to this process these pages are belonging to this process and there they are all you know disjoint sets of physical pages and so and you then create mappings appropriately alright.

So, let us assume that all processes are completely disjoint set of pages in physical memory and you are going to just set up your page table in such a way that the page table of process P 1 is going to point to which pages. And, process page table of process P 2 is going to point to which pages and each time you context switch you are going to overwrite the CR 3 register, yes.

Student: Sir.

So, for in the GDT, there is a u base and u limit for each process, yes ok.

Student: The segment register will point to that location; they will not be going to change?

Yes, all the let us say all the segment registers will always point to u base and u limit when the processes execute ok.

Student: So, that u base and u limit you should point to page directory?

That u base and u limit will not point anywhere ok. ubase and ulimit will only be used to compute a linear address right. So, from the offset, it will compute a linear address and now, the hardware what will point to the page directory is CR 3, is a completely you know separate register. You are going to use the CR 3 to actually look into the page directory, use the use the linear address computed through your segmentation hardware to index into CR 3, the first 10 bits and so on alright.

So, except that there is one thing, so this is this sound very similar to how it was done for segmentation right; the only difference really is in this case I am changing the page tables themselves. In that case, I was just changing the segment descriptor right.

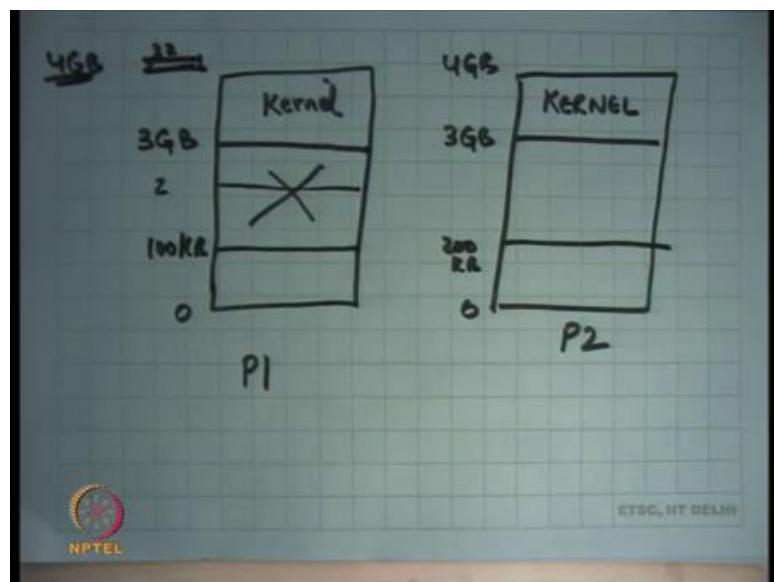
The difference here is that the kernel is also you know I said that in the in this model let us assume that the segmentation is completely flat. So, even the kernel's segment descriptor also has base equal to 0 and limit is equal to $2^{32} - 1$ right.

So, the kernel is also you know the kernel is also not using the segmentation hardware at all let us say right. In which case you know somebody else, so the virtual address to physical address translation is just going to take the offset and make that the linear address and now, the linear address has to be converted to physical address by a paging hardware right.

So, the kernel's linear address is likely it is possible that the, so the kernels linear address needs to be converted to physical address by the paging hardware right. And so, what you do is you inside the page table of every or inside the VA space of every P 1, you also mapped the kernel somewhere right.

So, the kernel shares VA space with the process ok. Let us see you know this, so basically let us take some concrete examples; how does it work in Linux alright. So, how does it work in Linux?

(Refer Slide Time: 28:25)



Every process has an address space which can go from you know by definition on a 32 bit machine, you can utmost have 0 to $2^{32} - 1$ except that you say that a process is not allowed to access the entire 0 to $2^{32} - 1$; if you are compiling a process for Linux or if a process should be able to run inside Linux legally, then it should never access an address beyond, so, you know this let me call it 4 GB just to make it more readable.

So, $2^{32} - 1$ is let us say a 4 GB and so, you should not ever access an address above 3 GB right. So, the process is constrained to access addresses only within 0 and 3 GB. The addresses from 3 GB to 4 GB, the virtual addresses are reserved for the kernel. So, that is where the kernel is mapped ok. So, let us say this is P 1.

So, typically what will happen is P 1 is mapped 0 to let us say some address you know 100 KB and all this is not used right and then, there is P 2 which also has 0 to you know let us say 200 KB this time and, but also has them kernel mapped from 3 GB to 4 GB right.

So, the kernel also ensures basically while setting up the page table that its own pages are always mapped in a certain address range of the VA space. So, if for example, there is a transition from user to the kernel and the kernel wants to access its own data structures, it can just use one of these addresses from 3 GB to 4 GB to start to try and to start and accessing its own code and data for example, because we did in this scenario we are not using segmentation at all.

What is happening in segmentation? Each time there was a switch from a user to kernel CS was overwritten and CS had a different base right and so, that way the kernel you know you are now actually accessing kernels memory and not-users memory immediately.

And you know the first thing the kernel will do is load other segment registers so that you know apart from the code other data, other segments are also pointing to the kernel. But in this model when everything is flat and the kernels also using a flat model, the user is also using a flat model, when you switch from when you switch CS, you are still in the same linear address space right and but you need you need to say that now I want to access kernel's data or and now, I want to access kernel score.

So, that the way it is done is cs the offset of cs is still 0, but the EIP there is certain EIP's that are reserved for the kernel. So, in particular the EIP is above 3 GB are reserved for the kernel on Linux right. So, in the interrupt descriptor table for example, the CS colon EIP, the cs base of CS if you are if I am not using segmentation the base of CS will be 0, but the EIP will be some address above 3 GB typically, that is where the handler will live right. And so, so because the offset is above 3 GB, the linear address will also be above 3 GB and now it will go through the paging hardware and you will basically go reach the kernels space.

It is the job of the kernel to ensure that the paging hardware translates the address 3 GB and above to the right place in the physical memory and that it can do by setting up the page table in such a way. So, each process is a page table, not only maps that processes address space, but also maps the kernel's the address space at the top in Linux alright.

On Windows you know this is typically the model followed in many operating systems, similarly in windows. Windows seem to you know sometimes need more space. So, it can actually take 2 GB, you know it actually takes the top 2 GB for its own kernel on the 32 bit machine, where it actually allows you to actually change it if you if you want right.

So, depending on so and this is part of the specification of the operating system. So, if you are compiling something for Linux, then you should ensure that your application will never access an address above 3 GB ok. So, it is part of the interface. The compiler should be aware of that or you know if the programmer is assembly programmer you should be aware of that. And how do you do protection? how do you ensure that the user is never able to access kernels data or execute kernels code?

Student: Sir, its flag.

The flag right; the user flag; so, the user flag is off for these mappings of kernel. So, for the all the kernel mappings the user flag in the page directory entry and the page table entry is 0; for all the other entries the user flag is 1. So, the user can never access that kernel. But if there is a there is a trap, then CS gets reloaded and the last 2 bits of CS will become 0 and now, it can access the upper regions of memory virtual address space question.

Student: Sir. So, these pages are common across those processes like (Refer time: 33:42)?

Right. So, just like we saw that in segmentation the kernel remains constant right, the kernel does not move the kernel in the physical address space remain sort of there and the mapping for the kernel remains constant. Similarly, the mapping this remains constant; so, the same entries which get copied in every page table or every page directory and page table of every process.

Student: Is it a kind of an exception for overrule that every page should like point to a different physical entry?

It is a yeah. So, it is a so, here is an example we have we are breaking the rule that every page table should be pointing to a distinct set of pages right. Here is an example where 2 different processes page tables can be pointing to the same page in physical memory. Yeah, great point and actually there are more examples which we are going to discuss later. Question?

Student: Yes sir. These kernel address space; does this also have the kernel stack for that process?

Does this have also the kernel stack for the process, the kernel address space?

Student: Yes, sir.

Yes.

Student: So, when the kernel stacks for two different processes, different from each other?

Right. So, let us look at how the kernel stack gets switched. So, we said that if there was a trap while I am executing the user mode, both the CS will get overwritten, and SS will get overwritten, EIP will get overwritten and ESP will get overwritten and so, what does the OS needs to ensure that the EIP is an address above 3 GB and the ESP is also an address above 3 GB and within this kernel space, you could be having multiple stacks; 1 per process.

So, actually you know in the kernel space you are actually having all the stacks of all the processes, all the kernel stacks assuming the process model of the kernel. So, in the kernel space you will have all the stacks of all the processes, but it is all inaccessible to the user. The user cannot access it right.

So, it says that the kernel has mapped its entire data structure and code into the user address space. So, if there is a trap and a flat segmentations model, then you straight away go to that to the kernel address space in both for EIP the program counter and the ESP the stack pointer ok. And, then you also change your other sort of segments and all that ok.

Question is why do I need the entire kernel to be mapped inside the process address space; could not I have just mapped the kernel stack in the process address space? Just one you know let us say whatever size I want to have for the stack kernel stack that is what I am going to put it in the, but what about the code? You know so the program counter, you will also need to put that. So, the handler needs to be there right.

And so, and the handler is going to make more calls. So, you could potentially say that look I do not you know this 1 GB of space taken away from the process seems too costly and so, what I want to do is actually have a very small sort of handler inside which is not 1 GB, but let us say you know few 100 KBs and that handler is actually immediately going to switch the page table and then, he is going to execute the kernel logic that is a possibility right. There are pros and cons to doing that. This seems this actually works this is 1 of the best performance kinds of designs.

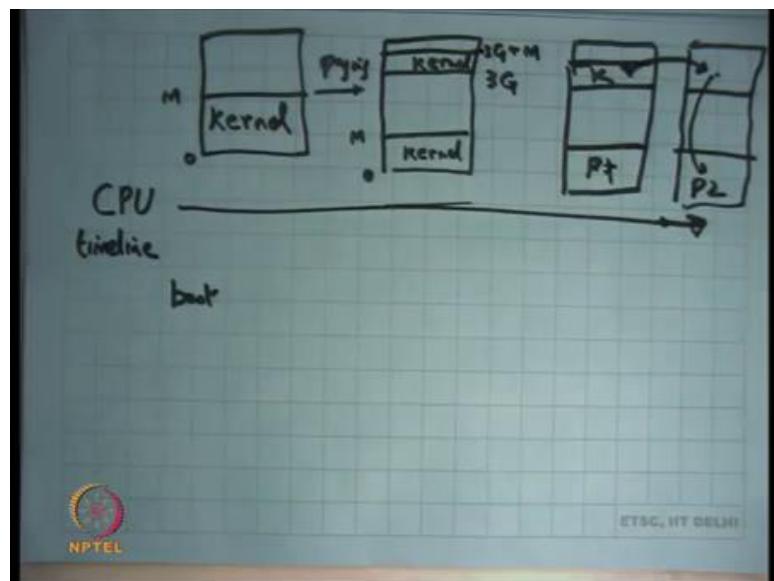
So, you know there have been multiple designs of operating systems that are being proposed, but this is this is sort of the most popular design and one reason this is the most popular design is it is the most performing design or I should qualify the statement. This is in a controversial statement may not be the most performing design, but it is the easiest to get performance out of this design alright and this is a design that you know your mainstream kernel use, Linux, Windows, x86 is you know what are you going to we were going to be starting next starting next time, on which base is going to be your programming assignments alright.

So, basically this is how. So, the kernel basically maps itself in every process address space and you know if there is a trap, it can immediately transition without having to do

anything and now kernel can now execute happily inside the process as possible. This is this is basically there is one difference that the kernel needs to be aware that it should only be accessing you know; it should be living in a certain address range in segmentation case.

The kernel could have mapped itself from 0 to something right because you know CS base has been changed. So, you know you are pointing somewhere else. But in this case the offset needs to be within a certain range 3 to 4 GB let us say on Linux ok. So, let us see what happens on a timeline just like we had seen for a let us say this is boot time, just like we had seen for the process.

(Refer Slide Time: 38:23)



So, what happens is you know at boot time it will first set up its segmentation table and now in this space, it will just set up the segmentation tables to completely identify mappings 0 and 2 to the power 32 minus 1. Then, it will set up its own page table which will be let us say an identity mapping and it will enable paging in the hardware right.

And so, it is the kernel page table that is getting loaded here; then, the kernel will load itself in a certain address space. So, the kernel has been compiled to assume that it will be living in a certain address range right. So, the kernels EIP and all these the program counters will always be in that address range and so, when it loads the kernel and the core kernel starts executing you know. So, let us say. So, initially you start executing in

physical address space. So, kernel is living from 0 to whatever maximum value you have.

Then, what the kernel does is it load itself from a you know while its executing, it loads itself in the top address space also. So, it first set up sets up page table. So, enabled paging it sets up the address space and now it says let us map myself in the new page table at the top.

So, it is going to map itself on the top also. So, it is going to say 3 GB to let us say 3 G plus M right. So, it is going. So, this is where it gets loaded and now, now it starts running from here and now it is you know context switch is to the first process, it starts the first process and so, the kernel lives here I want to say K and the process let us say lives here P 1 right.

And then, at again context which is when it context switch is it actually you know it executes. So, when a context switch is a trap occurs while it was executing here; when the trap occurs its starts executing here, here it does the appropriate changes in CR 3 in the page table.

So, when it does the page table change it actually reaches here. It has changed the address space its change CR 3 and now it calls return to get to presume execution of process P 2 alright. So, process P 1 is executing and user mode, a trap occurs, you switched to kernels address space in the same page table.

The kernel in that page table while its executing that page table, creates a new page tables for process P 2 if needed; maps itself in that new page table, switches the page table because it had mapped itself it exactly the same addresses when it switches itself you know it is not like the carpet had been dragged out under its feet because the same addresses are still valid right.

So, the new page table has the kernel mapped in exactly the same place where it was mapped in the old page table. So, when you switch it does not matter right. The kernels can still execute it just the user space address space that has changed and now, you can return back using the added instruction to the process and now the process P 2 can start executing.

Student: Sir, but kernel they got already a kernel space in the in the P 1 page table. So, why did you tend to do a context switch?

So, let us say you know the kernel has decided that it wants its context switch from P 1 to P 2 alright. So, it needs to load the new P 2 page table, but the nice thing is that P 2 page table also has a kernel mapped at exactly the same position. So, when it does, when it overwrites CR 3 its addresses are still valid. So, the next instruction pointer will exactly go to the same physical location still right.

Student: Sir.

And now the next thing it will do is basically a go to user mode and this time it will see P 2 there.

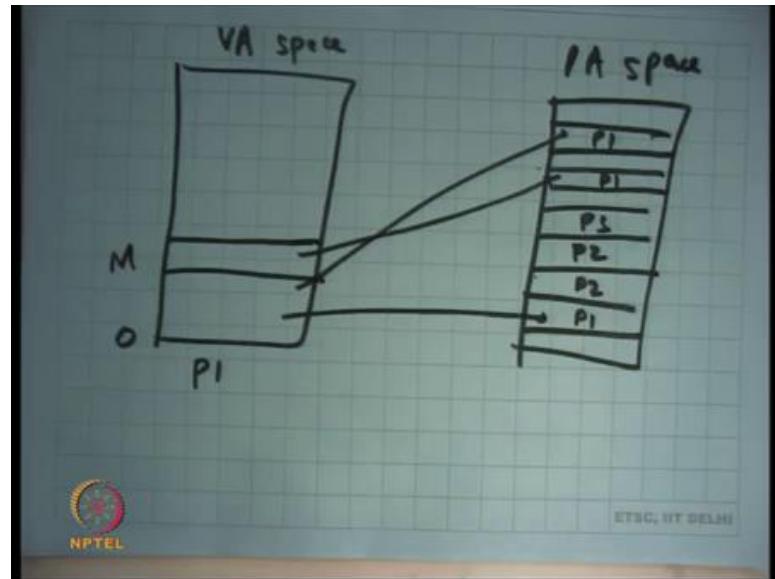
Student: Sir. So, the all the kernels spaces are each of the processes are exact replicas of each other?

All the kernel spaces in each of the processes are either replicas; well, they are actually pointing to the same place. So, they are not replicas, they are shared. They are not copies, they are just shared with each other.

Student: Sir, but page tables are replicas?

Page tables are replicas. So, the kernels space itself is not replicas. The page tables are replicas for those particular entries, not all the entries; just for those particular entries which are mapping the kernel ok alright. So, let us see some nice things that you can do with paging you can do something called. So, firstly, page paging solves the problem of fragmentation and growth in a large way right.

(Refer Slide Time: 43:23)



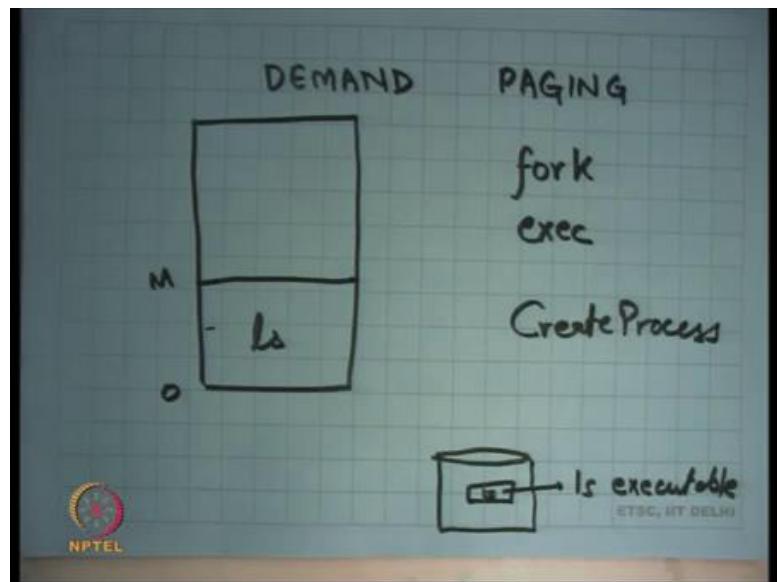
Because now if a process needs to grow all it needs to do is let us say this is the VA space and this is the PA space. So, let us say the process is mapped in 0 to capital M, but you know these are pages which are mapped here. Let us say it has 2 pages; one is mapped here and the other is mapped here.

So, let us say this is P 1. So, these are P 1's pages and these are P 1 pages and now there are other pages like P 2's pages P 3's pages. So, there is two; let us say two P 2 pages, one P 3 page and now there is all this empty space. Now, if I want to grow M all I need to do is you know create another mapping for another page, allocate another page here P 1 and create this mapping here right.

So, growth is very easy. No problems of fragmentation uh, you just sort of allocate a new page in the in the physical address space and you create a mapping in the page table. So, that the virtual address space grows ok.

So, growth is no problem; fragmentation is no problem because you know the mappings do not need to be contiguous, if there was enough space in the if there all they are enough pages for the new process to fit in, it will get fit in. It is not like they need to be contiguous alright ok. So, it is all fragmentation. The other nice thing it can do is what is called Demand paging right.

(Refer Slide Time: 44:51)



What is demand paging? Let us say I created a new process alright and let us say this processes ls right. So, let us say this is ls and ls maps its code in let us say 0 to capital M hypothetically and let us say there is this disk right which has this program file called ls executable right. ls executable.

Now, when you actually say ls, one way to implement the exit system call in the OS is to actually copy the entire contents of the ls executable into physical memory and create this mapping right that is you know up front you allocate that much memory, you copy the entire discontents to memory, create the mapping and start the process running.

And optimisation over that is that you actually do not read the disk into memory up front, you just create an address space from 0 to M by reading the executable and then, you basically just store some meta information in this address space saying that look right now these pages are not present, but if the user tries to access this page; then, here is where you should get it from on disk. So, here is the disk block from where you should get this particular page from right.

So, for example, if ls was a large program you know let us say the program was let us say 1 megabytes long large, but when typically when you type ls, you are only going to be executing let us say 100 kilobytes of code somewhere and you know almost you know let us it 4 kilobytes of data. So, you only needed those 100 and 4 kilobytes in memory, you did not need to read the entire 1 megabyte into memory.

So, what you do is you know demand paging helps, but the OS does not know you know which what may get executed in future. So, but demand paging helps it creates the address space for the entire 1 megabyte. It maps the first page of the page containing the instruction which is the first instruction into memory and transfers control to that instruction. If that instruction happens to execute some other address which is not currently mapped, what will happen?

Student: Page fault.

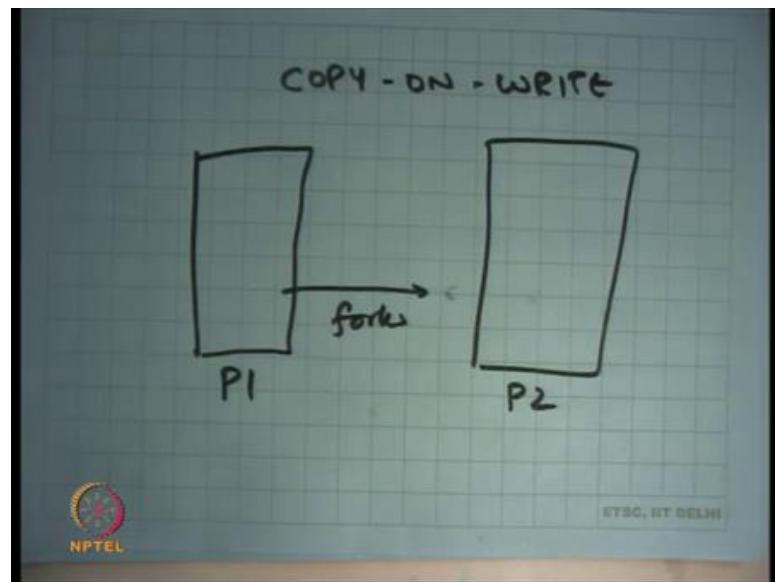
Page fault, the operating system will come into action, it will figure out oh you know actually the process is not doing anything illegal, its I who is playing tricks under the rug right and so, I should not I should actually you know stick to my contract and so, that is when you will basically pick up the page pasted into physical memory create the mapping and restart the instruction alright. So, this is this is demand paging, very useful optimization.

In fact, we discussed this we discussed fork and exec right in Unix and we said you know and then, we said oh windows has this create process right and we said look fork seems very base full because you are going to actually copy the entire process in from disk to memory and all that, but the reason you know Unix designers at that time thought that it is not a big problem. You can actually fork is not inefficient is because of demand paging.

At that time memories were really small. So, most of your program code and data use to live one disk right and so, fork was basically just fork forking a process just involved creating a new page table and you know updating and having the same pointer shared pointers through the disk and the fork was really cheap in that sense. And if the next thing the process going to do is exec you know no problem; you actually did not waste much work right. So, at that time it seems very natural and so, fork and exec was the nice interface given that demand paging was a very possible optimization.

So, I mean optimisations also dictate your programming models right, nice examples and another thing you can do is copy on write right.

(Refer Slide Time: 48:53)



So, what is Copy-On-Write? Let us say here is a process P 1 and it forks another process P 2 alright and let us say you know we are it is not the old world, in the new world then processes memories are large; so, the process actually living in memory right. So, once again fork is actually one way to implement fork is that you create a page table for the new process and you basically point the page table entries of both P 1 and P 2 to the same physical pages except that you mark all those pages read only right. Because if one process writes you do not want that to be visible to the other process, you mark all those pages read only. So, you so, the entire space is shared except that its now become read only.

Now, if one of those process types to write one of those pages, then immediately there will be a page fault and you will just copy it at that time. So, that is called copy on write right. So, you actually share the pages and assuming that you the child process was going to call exec immediately, its actually not going to write to any page right and the page table was going to get overwritten. So, fork is again very cheap even in the memory world alright.

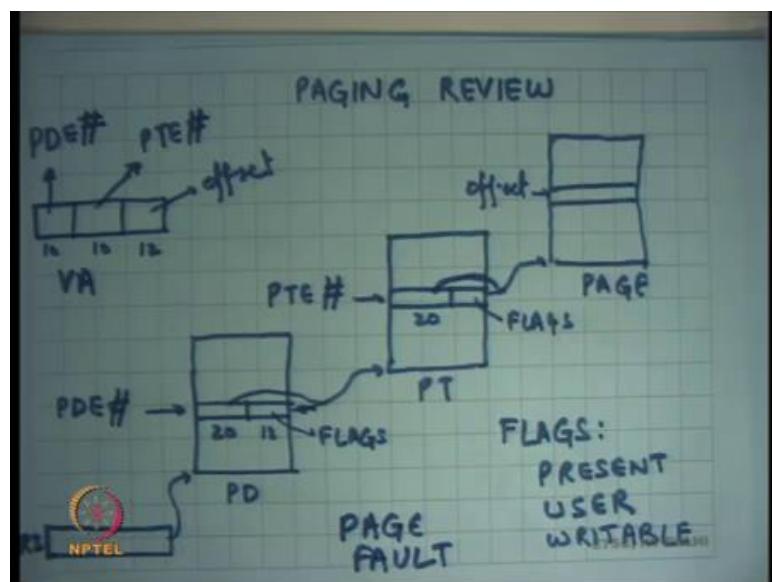
Ok good. So, let us stop here.

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

Lecture - 12
Process Address Spaces Using Paging

So, welcome to Operating Systems, lecture 12. So, far we had been discussing paging and we said that the hardware provides the capability of setting up a structure called a page table.

(Refer Slide Time: 00:44)



So, the operating system can set up the structure called the page table which we said that if we are using pages of size 4 kilobytes and an address space of size 2 to the power 32, then you would probably want to have a 2-level page table, so that it has a nice tradeoff between the time it takes to translate a virtual address to physical address and the space that such a structure requires, right.

So, we said a 2-level page table seems like a nice tradeoff between these two things. If you have just one continuous page table then translation time is very small, but at the same time the space over it is very large. If you use a multilevel page table more than two, then the space overheads can become even less, but then the size of then the translation times become unnecessarily large, right.

So, a 2-level page table seems like a reasonable tradeoff. And, so the way it works says there is a register on the chip, on the CPU processor chip called CR 3 which points to a page, a page sized structure, this is this entire thing is one page, but it contains its it is a special page because it contains page directory entry. So, this page is called the page directory.

The first 10 bits of the virtual address, so this is the virtual address; the first 10 bits of the virtual address are used to are called the page directory entry number and based on the PDE number you would see which entry you are going to dereference. From that you are going to get a 20-bit address which is going to point to another physical, which is be another physical address that will point to the page table, right. And then there will be the other 12 bits which are flags.

The page table is going to get dereferenced again in physical memory, so this is a physical address that goes, and you dereference a page table. You get the next 10 bits and that is basically forms a page table entry number. You use a page table entry number to look up the page table to get that entry similarly, and you look at the 10, first 20 bits to get the page number, once again this is a physical address, right.

And once again you have the last 12 bits which have been used as flags. And finally, you use the last 12 bits of the virtual address to index into this page to actually get the data that you were looking for, ok. I said that the PDE number is just a number between 0 and 2 to the power 10, right. So, they are 2 to the power 10 entries in one page directory because you are using a 10 bit number to do that and each entry is 4 bytes, so you know a 4 kilobytes, so the whole page directory fits in one page. Similarly, the whole, the one-page table fits in one page and the page itself is one page, right; 4 kilobytes.

Also, we said that this 20-bit number here is a physical address. Why does it need to be a physical address? Can it be a linear address?

Student: It is finally, (Refer Time: 03:38).

Right. So, it cannot be a linear address because linear address will need to dereference the page table to get converted to a physical address. So, if you use, if this were a linear address then I am basically in an infinite loop, right. So, that will, I will never be able to

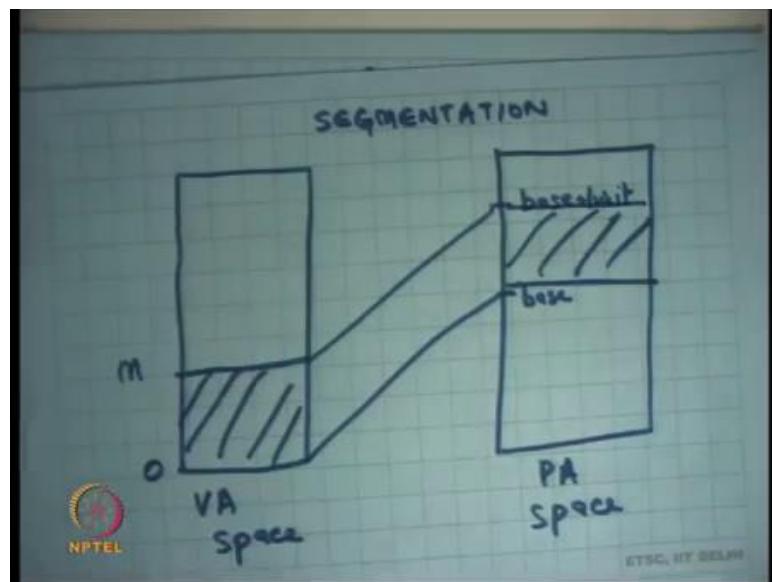
translate. So, this has to be a physical address, this has to be a physical address and so on.

What about CR 3? Should it be a physical address or a virtual or can it be a linear address? It should be a physical address also, right. It cannot be a virtual address. So, all these structures should be physical addresses, ok.

So, what this allows you to do is a much more flexible mapping from virtual address space to physical address space, right. Unlike segmentation where we had a very simple base plus VA computation, paging allows you to have a relatively much more arbitrary mapping except that if the mapping is done at page granularity, it is not done at byte granularity it is done at page granularity, right. So, you can say that this page is here, and that page is there.

But within a page two consecutive bytes should be consecutive in the physical memory also. So, within a page byte should be contiguous, both in contiguous bytes within a page on the virtual memory or in virtual address space are also contiguous in the physical address space within a page, ok.

(Refer Slide Time: 04:57)



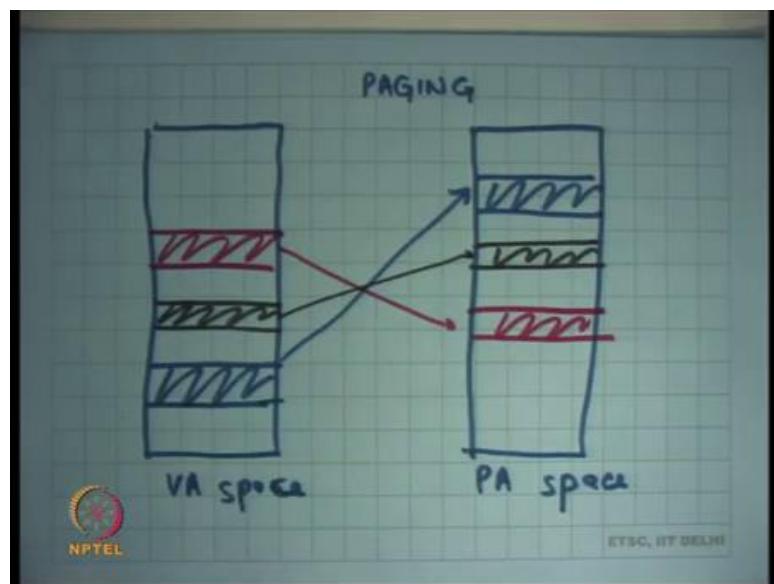
So, just to get a comparison here is you know how segmentation worked. If this was the virtual address space and its starts from 0 to let us say some value M, then segmentation would only be able to map it to some base plus 0 and base plus M, right or base plus

limit. And so, you know this is how the process will basically, that this is the only possible translation and we saw that this has some problems of fragmentation, process growth and things like that.

So, you know it is not very flexible way of doing it. The advantage however of doing things in this way is that the time it takes to actually translate a virtual address to physical address is very small, ok, it is very extremely fast. Just have to do an on-chip edition, right you do not have to look up into any structure nothing, right.

And we also said that the structure, the GDT itself although it lives in memory, it gets cashed on the CPU, right. And, when does it get cashed on the CPU? When you load the segment selector or a register. And so, at that time it gets cashed and later on you know it just used from within the chip, so the translation logic is really fast.

(Refer Slide Time: 06:04)



Student: Sir, in the case of paging sir is the address, this the base address of page directory also cached?

In the case of paging is the base is CR 3 cashed. We are going to talk about what gets cashed or not. But, by the way base directory of the CR 3, I mean CR 3 register is already on the chip, right.

Student: No like the address which CR 3 stores, the base address of the page directory.

That is on chip. CR 3 is a register which stores the base address of the page directory, right. So, it is already on chip. So, it is no problem. So, on paging, on the other hand you can have much more flexible mapping. So, here is an example of a VA space to PA space mapping on in paging.

Here I can you know do say that, these pages are mapped and by the and these pages are mapping to this area and these pages are mapping to this area and these pages are mapping to this area and all the other empty spaces are not mapped. So, all this is possible, right. I do not necessarily have to map from 0 to something, I can just say, or these pages are mapped, and these pages are not mapped.

And you know you can think about how you are going to architect your page directory and page table to be able to do this mapping, right. For example, if you have to do this mapping from here you are going to say, ok, what is this address? Let us say this address is M. So, I am going to say what does M divided by, you know M; what are the top 10 bits of M, right and those are, and based on that you are going to set up the page directory entry of that to point to a page table, right.

And, so now we are going to look at what are the next 10 bits of M and that particular entry in the page table should have a pointer to that page. Where? In this space. All other entries in the page directory and page table can be invalid, right. And how do you say whether a page table entry is invalid, or a page directory entry is invalid? Using these flags. So, if a present bit is not set in an entry, it basically means that this entry does not exist, it is invalid, ok.

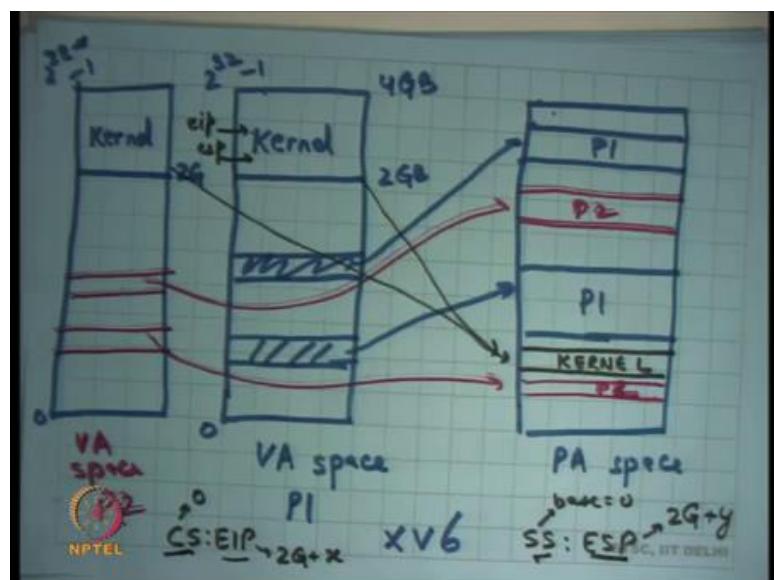
Similarly, there are other flags like user, basically says that this page should be, should be accessible and user mode or not, right. So, only if the bit is set to, this user bit is set in the flags is this page accessible while the processor is executing in a user mode, ring level 3, right. Other pages which do not have this bit set will not be accessible then user mode, they will only be accessible in ring level 0, kernel mode.

Similarly, you can have a bit which says whether this particular page is the page that is getting dereference from this PDE and PTE, page directory entry and page table entry is it writable or not, ok. So, if it is not writable then if some instruction tries to write to that address then you should get a page fault, ok. So, there is something called a page fault which is just an exception in the hardware.

So, an exception gets triggered and the exception handler got gets called, right and we have discussed how an exception handler gets called through the interruptive dissector, ok. So, that is what we discussed and then we said, let us look at how an operating system uses this page table hardware page and page directory hardware to actually implement processes, right.

So, what does an operating system really need? It needs a few things. It needs to say that my address space should be protected from everybody else's address space number one. Number two, one process's address space should be protected from another process's address space, right. Those are the basic two things that the operating system wants, and we have already seen how it is done in segmentation.

(Refer Slide Time: 09:53)



So, in paging one way to organize things and that is how we are going to focus on this way for the beginning, is that let us say this is the VA space, and let us say this is the PA space. So, one way to organize this is that you have per process VA space, so this is VA space for process P 1 and let us say this is let me use a different color. So, let us say this is VA space for process P 2, ok.

In the PA space you could say, here is, let us say this is a process P 1 page and let us say here is another process P 1 page and you can and you could have multiple of such regions, non-contiguous regions. And here is let us say P 2 page, see here is another P 2 page, ok.

And so what will happen is addresses in the P 1 page region it will map to areas in the PA space like this and similarly you know an entire this P 2, they are mapping here and so on, right. Also, we discussed last time that the kernel always stays mapped in the virtual address space and why it so, let us discuss it in a minute, but let us say how does it work.

So, how big is the VA space it starts from 0 and it goes all the way to 2^{32} minus 1, right on a 32 bit machine, but the VA space that is given to a process on something like Linux or let us say windows or XV 6, it is basically that you say a process will not get the entire 2^{32} space let us cut it somewhere, right and let us say that a process has allowed this size of the space. So, this 2^{32} is actually 4 GB, right.

So, one way to do it and let us say let us look at one particular operating system XV 6, right. So, what does XV 6 do? It says you know the process can use 0 to 2 GB and that is it, right. All the addresses above 2 GB in the virtual address space belong to the kernel, ok. So, the kernel maps itself in the address space of the process, except that this particular mapping has the user bit set to 0, which means these pages will only be accessible if the processor is executing at ring level 0, right in privileged mode.

These pages will not be accessible by the user if the processor is executing in unprivileged mode. And it does for every process. So, even in this process is a 2 GB, so exactly at the same place the kernel is mapped, ok.

So, a process has a page, has a virtual address space. Here it can have whatever it likes and those are getting mapped to different regions, potentially different region in the physical address space, but the kernel is mapped at the exact same place in every process. So, we will basically be saying the process and no process is allowed to use more than you know let us say 2 GB, one XV 6.

And of course, this is also blacked, so let us say let me use a different color for the kernel. So, black color and let us say let us say this is these are the kernel pages, ok. So, let us say these. Here is the mapping from the kernel and this stays constant. Irrespective of the process that is running, you will always have this mapping.

And so, what that means is, so how do you implement a different virtual address space for different processes? You have separate page tables for every process, right. So, every process has a separate page table and let us say if you are running process P 1 you will load process P 1's page table into CR 3, if you are executing process P 2 you will load process P 2's page table into CR 3, right.

But what will be common in these page tables is that there will be some entries which are for the kernel and which will always be there in all these page tables, ok. So, that is how you do it. You do not make copies of the kernel, the kernel and the physical memory is just one copy, but there are multiple processes which have this copy mapped in their address space, ok, all right.

So, why do you do this and let us see how it works basically. So, let us say I was executing in the process P 1 and some exception occurs, like a page fault occurs or let us say some external interrupt occurs, like you know a timer interrupt occurs or a disk interrupt occurs or a network card interrupt occurs. Then what happens is the processor goes through the IDT and replaces the current value of code segment and EIP with the value of the CS and EIP in the IDT, right and so typically these values will be the kernel values.

So, let us have with executing in P 1 and an interrupt occurs, then CS colon EIP that will be loaded from the interrupt descriptor table we will have CS is base as 0, right. So, we are assuming that the segmentation model is completely flat, so all segments are 0 to 2 to the power 32 minus 1. But EIP will be an address which is above 2 GB, right. So, EIP will be some you know 2 G plus x whatever, right.

So, it is going to point somewhere here. And so if an interrupt occurs or if an exception occurs the process straight away, the processor straight away jumps to the kernel address space, but it also reloads the CS, so that the last two bits of the CS register are set to 0. So, that you know in the processor is now executing in privilege mode, right. That is the only way it EIP can actually point to something in the kernel space, right.

A user if it just calls jump to this EIP, it will trigger a page fault, right because if the user is executing in ring level 3 which is the last two bits of CS, then it says jump to some address which is above 2 G immediately you know there will be a page fault because the

hardware will try to walk the page table and it will say you know you are running in user mode, but this particular page that you are trying to access is actually not a user page.

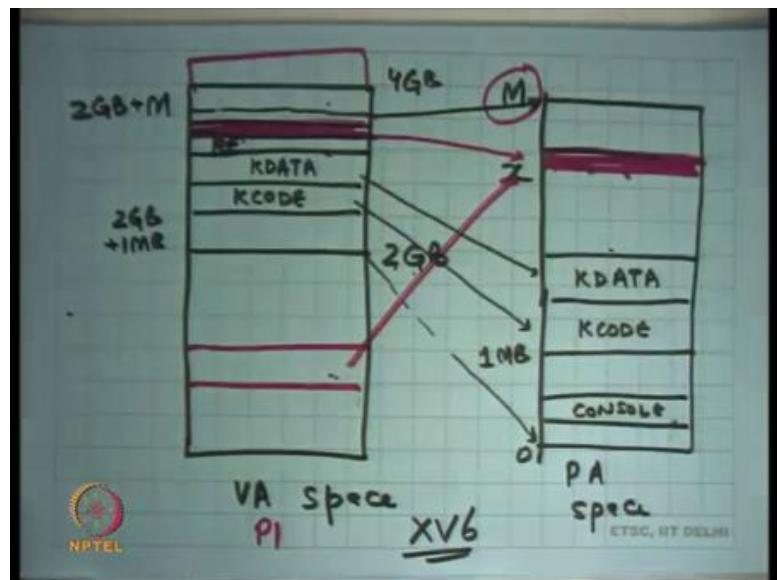
But if you get an interrupt like this then you also reload the CS, so now you are running in privileged mode, so now you can actually dereference a page above 2 G. So, the handler is now going to live in the kernel space. Also, the stack is now going to the kernel stack of the process is now going to live in the kernel space.

So, even in the SS and ESP, the SS's base is still 0 because we are using a splat segmentation model, but ESP values is some 2 G plus you know let us say y, so ESP is also pointing somewhere here, right. So, that is the kernel stack of the process that was currently running. Where do you get the ESP from? The task state segment, right. So, the hardware just reads a task state segment. So, with the responsibility of the operating system to set up the task state segment with the right ESP for that particular process.

If the kernel was a process mode kernel, right, it was the process model then you will have a separate ESP per process in which case you know before you contacts which to the new process you should load a new ESP in the TSS, task state segment. If it is an interrupt model kernel, then the ESP remains constant. This is the same stack that always gets executed, right.

So, in segmentation what was happening was the segment itself, the base itself was getting changed. In paging, the base remains same, the privilege levels changes, yes, but what is actually changing is the offset, which is EIP and ESP, right. And these offsets should be set up by the kernels at that they point to the kernel address space, which is 2 G and above, 2 GB and above, ok.

(Refer Slide Time: 18:53)



So, let us look at this again. So, let us say this is the, this is the virtual address space and let us say this is the this is 2 GB, all right and this is the kernel, right. So, the entire address space is 4 GB here. So, what typically happens is that you know the kernel will have some code and so you know let us say this is kernel code, KCODE.

These, this is this is the memory region which contains all the instructions that the kernel contains, right. And then you have certain region which contains KDATA which let us say contains all the global variables that you are that your kernel defined contains a KDATA and everything else above this is actually free space, right.

So, let us say this is your physical memory, so this is PA space and its going from 0 to let us say some value M capital M, right. And we said you know, the last time we discussed we said you know the addresses from 0 to 1 MB are actually this address space in the PA space is actually cluttered because you have some address range which is meant for the console, right for the VGA console, so if you write there it is actually not memory it is actually going to the VGA output, and the other things like BIOS ROM and all these things.

So, 0 to 1 MB on XV 6 it is a little cluttered. So, let us just leave it as it is, all right, but from 1 MB to M is space that is actually real memory that you can use with memory semantics. And so, what happens is you know let us say in this you would first load KCODE and KDATA. So, this is, and so this is physical memory.

And so, all the other space is basically what is available for other things, right. So, KCODE is mapping here and KDATA is getting mapped here, but all the other space in the physical memory is basically just extra space that you are going to use for other things. What are the other things? You are going to use it as a heap for let us say maintaining your PIB, process, PCBs process control blocks. You are going to use it as a space from where you are going to allocate the kernel stacks of the process, C's, right.

The kernel stacks of the processes are also going to get allocated in this space. Moreover, the address space of the process itself is going to get allocated from this space, right. So that, clearly this is the entire space which is going to get used for allocating address spaces for the process. It is going to be used for allocating the kernel stack for the process and it is going to be used for allocating the other kernel data structures like process control blocks or whatever else it needs, right. So, all that has to get allocated from this space, ok.

So, how, let us look at how XV 6 does this. It basically says, it says 2 GB is basically what the process will get in the VA space, above 2 GB is the kernel. Till 2 GB plus 1 MB, it is not going to touch anything, so it just maps this region from 2 GB to 2 GB plus M, where M is the size of the physical memory to 0 to M in physical address space, right.

So, 2 GB plus let us say this is 2 GB plus M gets mapped to 0 to M. So, the entire physical address space is also mapped in the kernels virtual address space, on XV 6. So, and so the kernel can now access the physical memory directly by just saying, if I if he wants to access physical memory byte number 10 or let us say any x then all I need to do is 2 GB plus x is then the virtual address space is basically physical byte number x, right.

So, let us say the kernel wanted to allocate an address space for the process, so what we will do is it will say let us allocate an address space for the process from here, right which will mean it will allocate an address space for the process from here. So, let us say let us use a different color. So, let us say it allocated an address space for a process from here which means it allocated an address space for the process from here, right and then it sets up a mapping in this area to this area, ok.

So, what is happened? See the, so this is the processes address space, right. Let us say this is P 1. So, P 1 wants to map some area of the physical memory and its virtual

address space. So, it is going to have a mapping in the physical address space like this, but this area is also mapped in the kernel address space like this.

So, here is an example where the same page in the physical memory is actually getting mapped both in the user side of things such that a user can access it, but it is also mapped in the kernel side of things so that when the kernel actually allocates it or deallocates it has access to all this area, ok.

So, basically just to recap, the kernel maps the entire physical memory in its address space. It allocates areas from this address space and maps it into the user's part of the address space to implement the processes address space or user side address space, ok. Mapping the entire physical memory into the kernels address space has advantages that you can just know if you want to access a byte number x all you need to do is look at you know your base plus x and you are get the particular address.

Student: Sir.

Yes.

Student: Sir, that address generally 4 GB, so how can you map 4 GB physical space into 2 GB?

So, question is how big is M? So, in this organization that I have discussed so far how big can MB.

Student: (Refer Time: 25:28).

At more, actually it is going to be 2 GB minus 1 MB, right because, actually yes 2 GB because you know 1 MB is also getting mapped here, so it can utmost be 2 GB. So, that basically XV 6 has this limitation that you cannot have more than let us say 2 GB of memory in your system, ok. Actually, I should also mention that the top few address spaces are also reserve for devices.

So, it is actually even less than 2 GB in XV 6. So, what does it; so, question is this? So, for if for something like XV 6 it is good enough number one, because it is just an academic operating system. But for something like a real operating system like Linux this may not be good enough. It may have been good enough in the early days of this

operating system like 90s when you know the memories were not, they were nowhere near 2 GB mark.

So, this was you know this kind of organization actually worked for hardware of that time, but how Linux deals with memories which are greater than you know let us say 2 GB and if you started at 2 GB is basically to recycle these addresses. So, depending on which address you want to access it just changes its address space on the fly, right.

Here I am talking about a static mapping. I am saying the entire memory is mapped in the kernel address space all the time, so it is completely static for the entire duration of the system that is running. But if you are actually running out of the address space because the address space of the kernel is only let us say 2 GB, but your memory is let's say 3 GB then you will say the first few MBs which contain all my handlers and everything those remain always mapped, but other things like the area from where I am allocate process memory for processes that I sort of keep recycling, right.

So, some part of this 2 GB slice will get recycled. So, sometimes its mapping to this area of the physical memory, sometimes it is mapping to that area of the physical memory and the kernel is basically ensuring that its always its doing what it wants to do, but some area of the kernel will always stay mapped because things like handlers, right or kernel stacks, they will always stay mapped.

The memory, the area from where a memory for the handlers is taken which is KCODE should always remain, right you cannot just remove it at any time. Similarly, the memory from at which the kernel stack is living for that for the current process that should always remain and some other things which are basic functionality of the operating system they always remain in the kernel.

Student: Sir, if our physical address space is growing then why cannot we just grow our virtual address space?

If our physical address space is growing why cannot we grow our virtual address space? All right. So, question is if my physical hardware memory is growing then why I cannot just switch to a 64-bit system, that is what; that is what has happened, right. So, one of the big motivations is actually moving to a 64-bit system is so you get rid of these kinds

of limitations, ok. But, let us just focus our intention on 32-bit system because you know that simpler and will help us in understanding many concepts, ok.

So, yes, if you had a 64 bit address space then you can imagine that these are absolutely no problem, non-problems, right because there is no way a physical address space today at least or actually anytime in foreseeable future that you can have actually a memory which is 2^{64} bits long, right that is I think you know I do not know how many items are in the universe, so probably its comparable, ok.

So, this is the organization which let us say XV 6 users and in fact, other operating systems also use. So, let us understand what the advantages are of doing this kind of organization. So, the entire kernel space is mapped in every process and we said that the entire physical memory is mapped in the kernel space.

So, essentially the entire physical memory is mapped in every process in the kernel side of things. And then there is a user side of things which is controlled by the kernel, and so anytime what this gives you the kernel is that if there is an interrupt while the user is executing you do not have to change any page tables or anything you just switch the privilege level and you are executing in kernel mode.

If you want to do some operation on behalf of the user for example, you let us say the user said I want more memory, so all I needs to do is allocate something from its own address space, convert it into its corresponding physical address and create that particular page table mapping, right. So, it can only use, it can use its own malloc function inside the kernel to manage that space the that it has which is actually the entire physical memory, right.

So, it can just use malloc to create a page, let us say it wants to allocate a page for the physical process for the process then it just create mallocs a page, converts, it gets its address, converts that address to the physical counterpart, creates the corresponding mapping in the page table at the, right position and that is it, returns back to the user, right.

So, any interrupt or exception or system call does not need any change in the page table. So, it is very fast in that sense, because the kernel is mapped entirely in the address space of the process although in privilege mode it sorts of improve things in that sense, ok. The

other thing is let us say you have a system call and this system call will have some arguments, right.

So, things like exec and exec is a system called which takes the first argument as string. The question is, how does a process give an argument to a system call. How does a process; how does the argument of a function work?

Student: (Refer Time: 31:16).

Right. So, if exec what the function call, the pointer to the string which holds the name of the executable will be pushed on the stack, right and the stack address will be visible to the function, right. So, the function can just look at the stack address and it can dereference at address and that the dereference will also be visible to the function because that address to which it points will also be mapped in the user address space, right.

So, it can dereference the argument and it will still get a value which is in the user address space which you can access, right. Similarly, in the case of a system call the user can use a similar organization, it can store the string in its own address space, for example, its slash bin slash ls and then make a system call called exec. The control moves to the kernel, but if the kernel wants to read the arguments you are still executing in the address space of the process.

So, it can just dereference of argument and because the page which contains that strings slash bin slash ls is still mapped dereferencing should still work, right. The kernel would be executing in privileged mode, but privileged mode execution can access unprivileged pages. So, it can just, so accessing the arguments of the user is very easy. You just have to dereference the pointer and you can just get the arguments, ok.

Once again, there is an advantage of mapping the kernel into the user address space because if the kernel is doing something on behalf of the user and the user wants to give some pass some information to the kernel it is very easy to pass information from the user space to the kernel space because they are both living in the same address space. The user can just set up something in its own address space and give a pointer to the kernel, right. The kernel can just dereference a pointer and it will be able to read the value because you are in the same address space.

Compare this with segmentation, right. In segmentation when there was a system call then you were actually changing the base. So, now, if the user wanted to give an argument to the kernel, I cannot, the kernel cannot just dereference a pointer because you are in a new address space, the base is different, right. So, the pointer that the user gives you, if you just dereference it now you are actually going to be looking at something else, maybe something wrong, right.

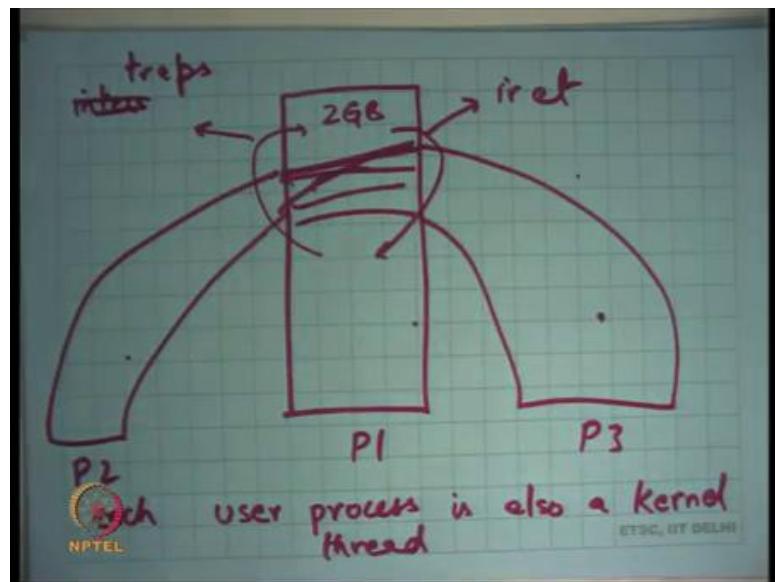
So, in the case of segmentation what is actually needed is that this the user actually, user needs to you know package all its arguments and copy it into the kernel space somehow, right because the kernels address space and the user address space are separate and so you need to copy it into the kernel using some kind of a mechanism. And in this case, you can just simply dereference the argument, right. So, that makes things faster.

Student: Sir, in segmentation could not the user end pass its own CS for inter as well for giving arguments, if it gives both the?

That is an interesting question. So, could not in segmentation could not the user have you know could not the kernel have changed through. So, the kernel already knows what the users' segments are, right. So, the question is cannot you, in the in segmentation cannot the kernel just switch to the user segment before dereferencing the pointer, right that is your question.

So, but that requires you know switching back and forth, and it will basically the, so that is this that is how it will typically be done, but that is requires switching segment, segments and then copying. So, typically what you will do is you will switch to that segment, copy it to your own segment and then switch back and then start operating on it, ok. So, there is more overhead in that sense. Question, ok.

(Refer Slide Time: 35:09)



Let us review this once again. Here is another way of drawing this, right. Let us say, so this is process P 1, this is process P 2, and this is process P 3. Each of them has a separate address space in the lower 2 GB, but they have the other same address space in the upper 2 GB, right. And what that means is if process P 1 makes any change in the kernel side then process P 2 will see it immediately because they are sharing the address space at the kernel side, right.

So, another way to say this is that the user processes, each user process is also a kernel thread, right. Recall, what is a thread? We said a thread is basically you know, threads are basically control flows which share an address space, right. So, these are controlled flows that are sharing an address space, in this case the address space of the kernel, right.

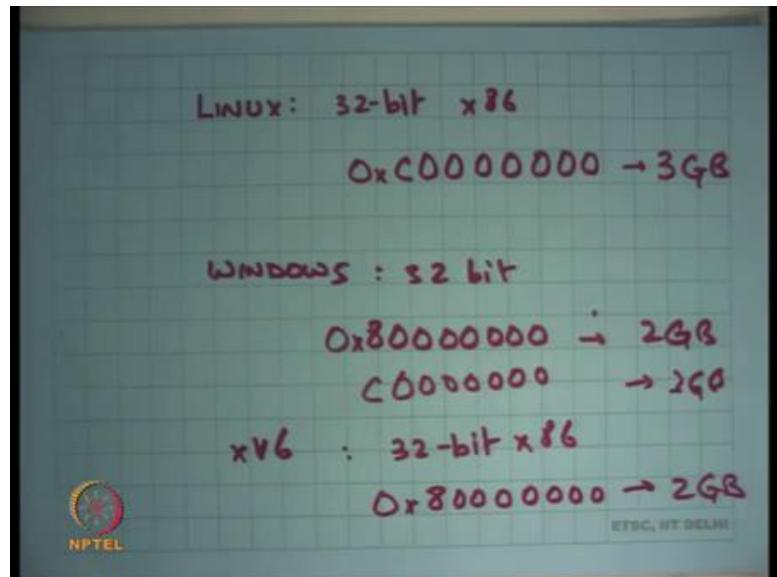
So, even though there are processes at the user level because they have different address spaces at the kernel level in this organization, they are threads, right. So, each process has a kernel half and a user half, and it switches between the kernel half. So, how does it switch from the user half to the kernel half?

Student: Through interrupts.

Through interrupts or exceptions or let us call them traps, ok, that includes system calls. And how does it switch from kernel half to user half? Let us say using the iret instruction, right. And so, but whatever change one has made the other one will see

immediately, ok. So, in that sense it is a thread. They do not have separate address spaces, ok.

(Refer Slide Time: 37:24)

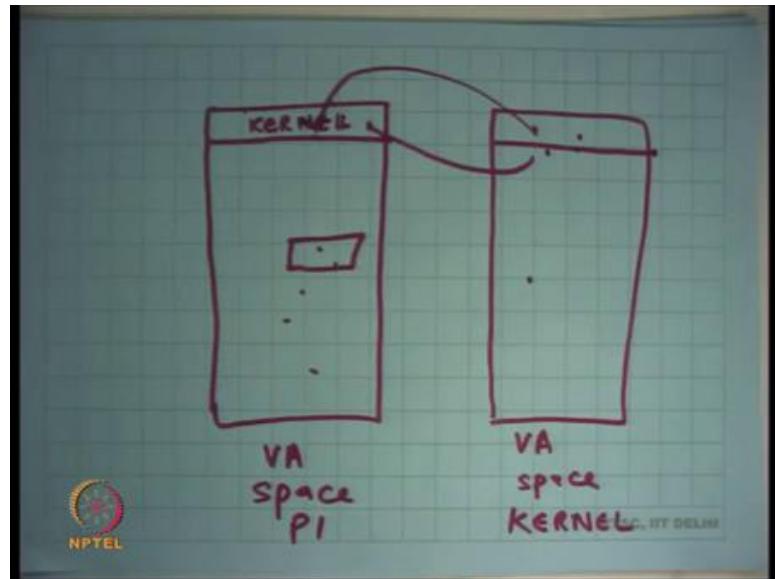


Just some facts. Linux on 32-bit x 86 starts the kernel address space at 1 2 3 4 5 6, at this address, which is 3 GB, all right, ok. Windows on 32 bit, I think by default starts at this address that is 2 GB, but you can configure it to say that you know please start at 3 GB, all right XV 6 on 32 bit x 86 starts at 2 GB, ok.

So, all these operating systems are actually mapping the entire kernel into the address space of the process and basically because it improves, it allows very fast switching between user and kernel number one you do not have to change any address spaces. Number two, it allows very fast communication from user to kernel, for example, arguments and maybe even return values, right.

So, very fast communication bit back and forth between user and kernel because at the same address space. All you need to do with dereference a pointer and you are there. Let us for completeness also understand what could have been the other organization, right. Let us say if the entire kernel was not mapped into the process address space what is the other possibility, right.

(Refer Slide Time: 39:15)



The other possibility is let us say this is the virtual address space of P 1. It just maps a small area for the kernel, right. And let us say there is another virtual address space which is just the kernel's address space not associated with any process.

And so what will happen is if you are executing in process P 1, the handlers and all that are still in this space, so if there is an interrupt you will get jump here, but the first thing the handle I will do is switch page table and you are here, right. And if you want to go back to the user, you are going to switch it back and then you are going to go back to the user using iret, right. But notice that this involves number one, switching page tables on a kernel execution, number one.

Number two, if I want to communicate from here to here it is not easy because what I need to do is copy my string from here. Let us say I wanted to communicate a string. So, I need to copy the string from here to here, then he needs to copy it from here to here, right, so the way he will copy it from here to here is let us say this same area is also mapped here, right. And so, if he copies it from here to here it is immediately visible here and so when it switches it, it now you can see it, right. It involves extra copying basically from user to kernel. So, it is a relatively slower.

Student: Can we say it is same as segmentation?

Can we say it is same as segmentation? Yes, roughly same except that in segmentation you did not even, I mean segmentation is giving you a mechanism, it does not even need you where you do not even need to map this area into the user address space because it is a completely different segment, all right. Question?

Student: Sir, sir in the kernel region all the addresses are linearly mapped to PA space, physical address space.

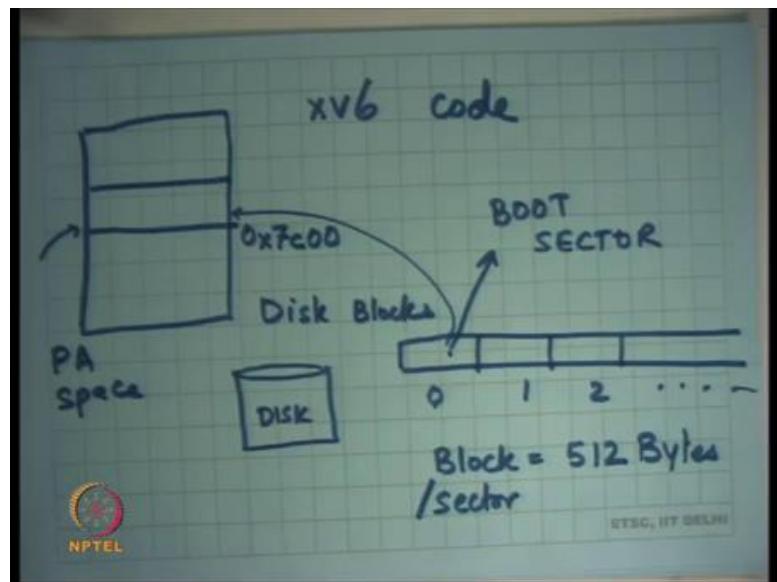
Yes.

Student: Sir, so we do not even need another this table it another table, we can directly just know that this is training in a kernel mode, so we can directly?

So, question is you know if the kernel is just going to have a one on, one to one mapping from virtual address space to a physical address space why do I need another VA space? Well, just that you know once you enable paging on hardware you cannot just access physical address directly you have to go through a page table.

So, once you enable the hardware you know every address will go through the page table. So, if you want to access the physical address you just set a one to one mapping from virtual address space to physical address space, all right. So, that is you know that is my, that is the introduction to paging. And from now onwards we are actually going to look at XV 6 code and look at XV 6 code to understand exactly how it is doing all these things that we have so far discussed, right.

(Refer Slide Time: 42:14)



So, let us just understand, so we are going to start with how a computer boots right. So, how does it appear to boot? Across power cycles you know if you switch on, switch off the power and you switch it on everything in your memory it is just everything gets wiped out, no processes, no kernel space, no user space nothing possessed. The only thing that possessed is a disk, right.

And so, when you boot on the computer there are certain conventions on how the disk should be laid out for the computer to actually start, right. So, let us just; so, a disk is actually a cylindrical device, but let us just consider it as a linear address space of blocks, right. So, this is block number 0. So, disk blocks. So, block number 0 and block number 1, 2, right and let us say it is very large, ok.

So, that is how the disk is. It is a collection of blocks where each block and once again this is the property of the hardware is 512 bytes, right or sector, also called sector, all right. So, each block each sector or a block is 512 bytes and what happens. So, once again the architecture defines exactly how it is going to interpret the contents of the disk.

So, the first block of the disk which is block number 0 is called the boot sector, right. So, and so when you start up your computer, the first thing let us say this is physical address space, right. So, the first thing the computer will do before even the first instruction of your operating system executes is that it is going to load the sector number 0 into some

predefined address, ok. It is going to load it here and it is going to transfer control to the first instruction in that sector, right.

On x86 this addresses hexadecimal 7c000, ok. So, you load this particular sector of 512 bytes, at this address at this physical address 7c00 and you transfer control to the first instruction. Recall, from a previous discussion that when the computer boots on x86 it boots in 16-bit mode, right.

So, in 16-bit mode there is no paging, segmentation is you know of a very primitive form where you just multiply the base with the segment value with 4 with 16 and added to the virtual address. So, there is no GDT or anything; that is what. So, at this point the processor will be executing in 16-bit mode.

The boot sector should be what should the boot sector do? So, the hardware will only load the boot sector from here to here. The boot sector should know where the rest of the kernel is and now it should load into the right memory and transfer control to it. So, in those 512 bytes the kernel developer needs to write code which loads the kernel into some other space and then jumps to it, right.

Fortunately, 512 bytes are enough to do this small operation, right. Because what you are going to look at next time is basically the code to do this and then what does the kernel do next and next and next, right and so how does it (Refer Time: 45:54) paging and then how does it starts the first process, ok.

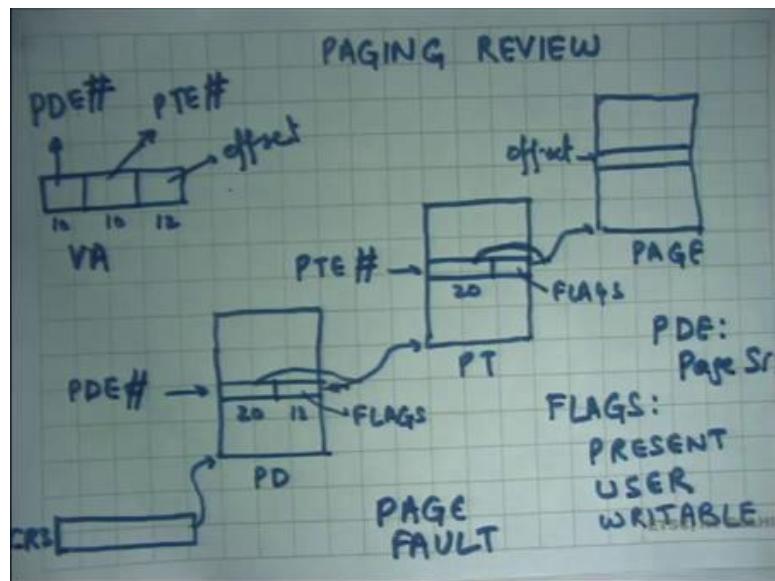
So, let us stop here.

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

Lecture – 13
Translation lookaside Buffer, Large Pages, Boot Sector

So far, we had been looking at paging and let us review it once again.

(Refer Slide Time: 00:09)



We said that segmentation allows you mapping from virtual address to physical address, but it only allows contiguous mappings because it is a simple base plus VA gives PA mapping which is not very flexible. It does not it has problems of fragmentation and it has problems when if a process wants to grow etcetera.

So, if you add a more general mapping and that is what paging implements, then it would be much more flexible. And, what paging does is divide the physical address space and the virtual address space into page lined, fixed size, page sized units right which are called pages. So, aligned fix size units which are called pages and, now, there is a mapping hardware in the middle which will map a page in virtual address space to a page in physical address space.

This mapping hardware is basically requiring a table which can have at most you know 2 to the power 20 entries. So, such a large table cannot be stored on chip. So, the such a

table needs to be stored on physical memory right or you know the technology for physical memory is also called DRAM.

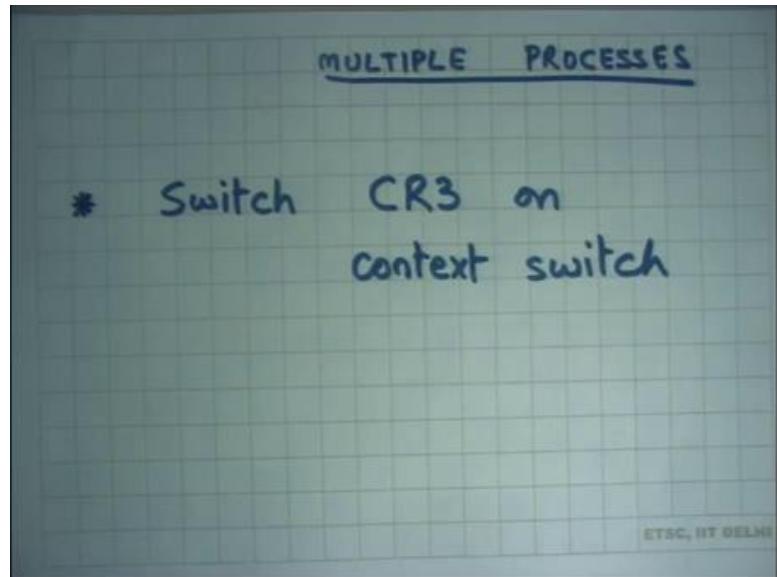
So, you store the table in physical memory also we said that rather than storing the table contiguously which will make it very large 4 megabytes; let us divide it into a two-level hierarchy, and so, that way small processes will only have a very small page tables and large processes will also you know will also benefit in space. Even if there are all the pages mapped, even then you know the two-level hierarchy is not too much space too much space hungry than hungrier than one-level page hierarchy alright.

So, on the chip there is a register called CR3 which holds a physical address, which points to the base of the page directory. The virtual address is used the 10 bit out of the 10 bits of the virtual address are used to index into the page directory from which you get the 20 bit physical address of the page table and 12 bits of flags. These flags are you know three use three relevant flags for us are present, user and writeable. So, whether the page is present, whether page table is present in this case, whether user mode execution can access it and whether it is writable right.

So, for example, if this is non-writable then the all the pages which are reference from here, they are all non-writable right. On the other hand, if this is writable then depending on what the flags are for the in the page table corresponding page table entry the page will have writable or read only permissions alright.

So, the top 10 bits are used to index the page directory, the next 10 bits are used to index the page table, and the last 12 bits are used as an offset into the page alright.

(Refer Slide Time: 03:02)



And, we said that an operating system can implement multiple processes or multiple address spaces by switching the page table on every context switch right. So, each time the process wants the operating system wants to change from P1 to P2 you change the CR3 value from P1 space directory to P2 space directly. You just load a new value into CR3, and you have a new page table.

And, we have also said that the kernel maps itself into the address space at the same point in every page table. So, a process does not see the entire 4 gigabytes of virtual address space, it sees something less than that. On xv6 it only sees a bottom 2 g gigabytes and the top 2 gigabytes of the virtual address space are reserved for the operating system or the kernel. And, so the kernel is really mapped at the same place pointing to the same physical addresses in every process right.

And, so we also said that every hence every user process is also a kernel thread right because a process now has two halves. One is the user half which the user can access and the other is the kernel half which only the kernel can access. A user cannot just switch into the kernel half the only way a user can switch into the kernel half is through the interrupt descriptor table through a trap right.

And, once it is switches into those kernel halves then it executes in kernel mode. It is possible that multiple core processes simultaneously are executing in the kernel half in which that is why we are calling them kernel threads ok. On the other hand, if the

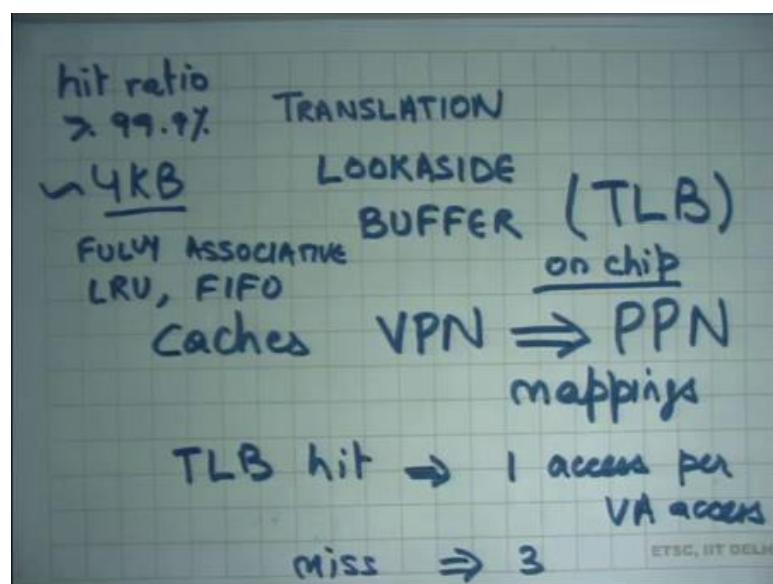
multiple processes are executing concurrently on the user half no problem because these are independent address spaces alright. So, how many accesses memory physical memory accesses does the processor need to make on a memory accessed by the program?

Student: Three.

Three. So, first you will dereference first you will add these 10 bits through CR3 and then dereference this value. So, that is memory access number 1. Then, you will get these bits and dereference this value – that is memory access number 2 then you will get this and then dereference this value, that is 3 right. So, for every memory access you are actually making three physical memory accesses. So, for every virtual memory access you are making three physical memory accesses and that is not acceptable right that is too costly.

In segmentation, we saw that such a problem was solved by basically ensuring that when a segment descriptor gets loaded the segment descriptor gets cached into the chip right. So, you do not need to access memory to get the base value each time; base value is just cached on chip.

(Refer Slide Time: 05:50)



Similarly, in paging, also there is caching involved and there is a special cache called the translation lookaside buffer translation lookaside buffer or TLB for short ok. This cashes

VPN virtual page number to PPN mappings. So, each time you basically access a page, a virtual address you walk the page tables the 2 two-level page table and you get a physical address right. But there was a mapping from the virtual page number to the physical page number that this page table structure was providing.

So, if you could cache this end-to-end mapping from VPN to PPN in your TLB then and you could first check your TLB and so, TLB is on chip right. This is on chip, on CPU. So, if the virtual address the VPN corresponding to the virtual address being accessed is already in the TLB, you do not need to walk the page table, you can just get the VP PPN from there add the offset and directly access the memory.

So, that way you just have to have if you get a TLB hit implies 1 access per VA access right 1 physical memory access per VA access. A miss implies 3 right. So, after the TLB miss you have to walk the page table just like before; if the TLB hit you do not need to go to the physical memory right.

So, it is important in real world that most of your memory accesses are TLB hits. You do not know most of your virtual address accesses are TLB hits for to have any acceptable performance right. Otherwise you know the whole idea of paging falls flat if the TLB hit ratio was not high enough right because segmentation was giving me a very fast translation. If paging is giving me a 2x overhead on every memory access that is not; that is not acceptable right.

So, fortunately most programs have a very high locality of access. Especially, because we are dividing the address space into page level granularity one entry one VPN to PPN entry is capturing locality of access within an entire page right. And, so it is possible to have a small TLB and yet have hit rates of 99.9 percent or above right and that is the kind of hit rate you will want in such a system to have any acceptable interrupt acceptability of this paging idea right.

So, you know typical TLB sizes on modern processors will be let us say 4 kilobytes also right roughly. So, you know a 4 kilobyte TLB can cache let us say; let us say each TLB entry requires 8 bytes, then a 4 kilobyte entry will 4 kilobyte TLB can cache 512 such pages page entries and assuming that your memory footprint. So, 5 so, you can basically cache 512 VPN to PPN mappings in your TLB and assuming that the programs memory footprint is less than 512 pages, then pretty much you should be able to access you know

this translation should be pretty much free. You are only accessing the on-chip TLB alright.

Also, let us understand whenever we talk about caching so, assume you have seen caching and you know what is hit ratio and all that right. So, I have been talking about hit ratio and let us say this is you know greater than 99.9. Let us say, now, just a ballpark figure 99 to 100 percent somewhere in the middle and that is the kind of hit ratios you are looking at ok.

So, when you talk about caching we should also say you know what is its cache replacement policy; what is its associativity; what is the whether it is a write through or write back cache right. So, you have seen all these terms in computer architecture class right. So, firstly, what is its associativity which means you know you know about cache associativity.

So, typically the TLB is a fully associative cache right you want your TLB to have such a high hit rate, so, you better make it fully associative. If on the other hand you had a direct mapped cache you will just have conflict misses in your cache right ok. The other thing is what the caches replacement policy. Well, the hardware can is free to implement any cache replacement policies it likes. Given that most of the things are going to be hits.

You know cache replacement policy does not cache replacement policy usually matters if the size of your caches are small. If your cache size of the caches are bigger than the working set size, then cache replacement policy is actually not that crucial. It does not really matter what your cache replacement policy is.

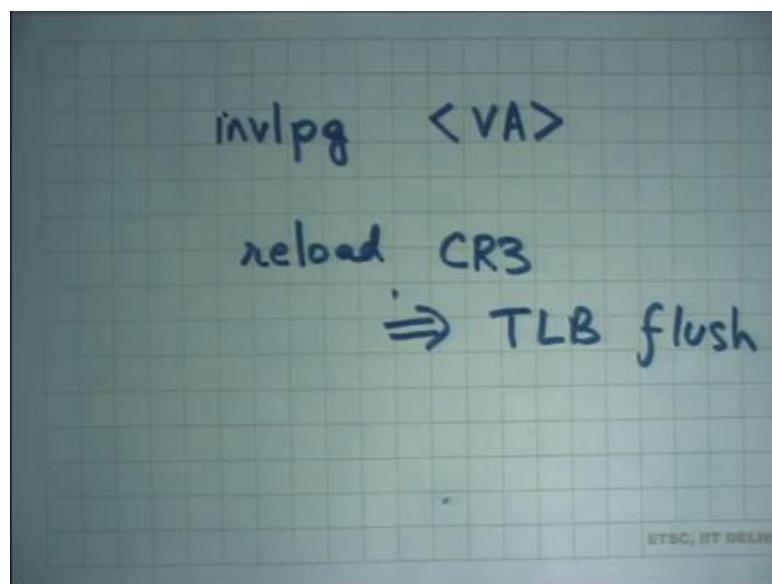
So, you know you can use something like LRU which will require some bookkeeping or simpler policy maybe FIFO and the hardware designer can make it make a choice you know. FIFO is simpler to implement, but it may have a slightly lower hit ratio than LRU. LRU is more complex to implement. It may or may not have better hit ratios than LRU you know that.

Now, let us look at whether it is a right back or a right through cache, let us see what happens. Each time a page table is walked by the hardware the entry gets cached into the TLB. So, this is a read operation, right? All these lookups are just reads of the page table.

The only right to the page table is then the kernel actually overwrites some entry by looking at that particular address right.

Recall that the page table itself is mapped in the kernel address space right. So, the kernel can change the page table by just writing to a memory location there. If the kernel changes a page table entry TLB does not even come to know about it right. So, a kernel needs to space the kernel needs to tell the hardware explicitly to invalidate a page directory entry or TLB entry right.

(Refer Slide Time: 12:59)



So, there is an instruction called let us say invalidate page and it takes a virtual address right; basically means invalidate any entry in the TLB corresponding to this virtual address right – so, explicit invalidation of the TLB entry, the cached entry. If the kernel forgets to do that, it is a bug right. Very bad things can happen ok.

You can imagine what can happen. What can happen is that the kernel thinks that it has mapped a certain page somewhere else and removed the entry here where the TLB still caches that entry and, now you know a user can probably access somebody else's page right. So, those kinds of bad things can happen.

So, it is important of the kernel executes each time it changes a page table entry it invalidates the virtual the page all TLB entries corresponding to a particular virtual

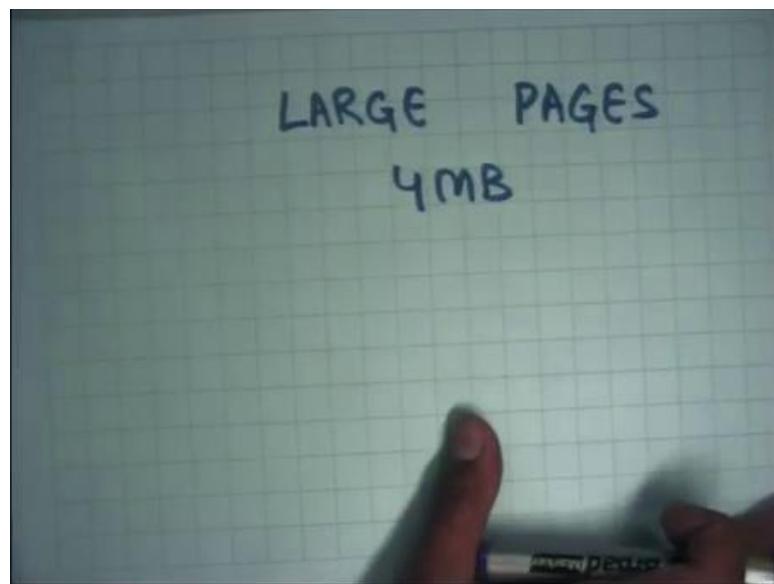
address or the single TLB entry corresponding to virtual address. Also, each time you do a context switch you change CR3 right. So, what should happen to the TLB?

Student: (Refer Time: 14:04).

All that you know the entire address space has changed right. So, you should completely flush the TLB right. So, reload of CR3 implies TLB flush right. So, in every context switch you flush the TLB completely because you know the entire address space is changed all the VPN to PPN mapping have only cache VPN to PPN mapping need to be invalidate alright.

So, it is important for an operating system to ensure that the number. So, the number of entries in the TLB typically remains small you know and one way that one method that the hardware provides to do that is supporting what are called large pages on x86 right. So, let us see.

(Refer Slide Time: 15:09)



There is something called large pages ok. So, we said you know normal pages are 4 kilobytes, but large pages are 4 megabytes. So, you could have large pages and how would you do the x86 architecture implement a large pages? In the flags of the page directory entry there is yet another bit. So, for PDE flags there is another bit which says page size alright. If the page size bit is 0 it means it is a normal page; if the page size bit is 1 – it means it is a large page.

And, if it is a large page it treats the 20-bit pointer not as a page table, but as a pointer to that 4 MB page ok. So, if it is a large page then this pointer is pointing to a 4 MB page directly right because you can imagine that you know a large page if your machine is talking about a 4 MB page you have divided your physical address space into pages of size 4 MB right.

And, so pages of size 4 MB means 2 to the power 22 bits I will basically use for an offset right and the top 10 bits; 22 bits are used for the offset and the top 10 bits are used to identify the page number right. And, so the top 10 bits of the virtual address are used to index the page directory entry to get the large page number right. Even large pages need to be aligned at page granularity right.

So, large page cannot, a large page just always starts at 4 MB boundaries right. So, there will be a large page at address 0 and then the next large page will be at address 4 MB and 8mb and so on; just like small pages were aligned at 4kb large pages are aligned at 4 MB right. And, so, 10 bits are enough to name a large page and so, the top 10 bits of the virtual address are enough to name a large page. And, so, the page directory entry directly points to a 4 MB region right.

If you are using small pages, then you allow this 4 m in any case the page one entry in the page directory is quite capable of mapping 4MB of virtual address space. If you are using small pages, this 4MB can be fragmented in 4kb chunks across the physical memory if you are using large pages then this 4MB chunk has to be contiguous in physical memory right that is the only difference. So, what is the advantage of doing this?

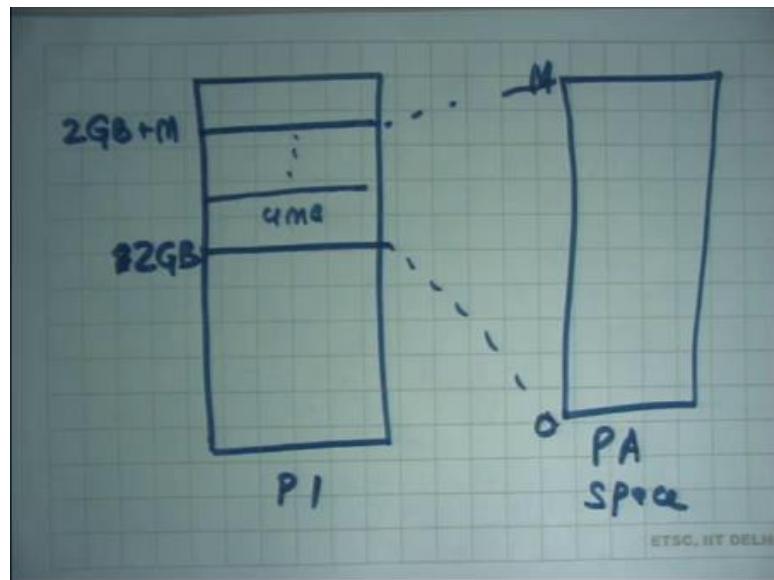
Student: Time.

First advantage is the time it takes to actually walk the page table has reduced. You only need to dereference one, but you know make two accesses for one even if there is a TLB miss. More importantly the number of entries that need to be cached in the TLB has reduced alright. As supposed to 2 to the power 10 entries for a 4-megabyte space you need only one entry, if you can ensure that the entire space is contiguous.

But it also means that the program should you know the operating system should take care of things like fragmentation right. For example, small pages small processes should

not be given large pages. Large processes can be given large pages, but in make sure that they are actually using those large pages because an entire address space, if they are not then I am basically worrying about fragmentation and all that kind of issues. One big place where large pages are really useful is to map the kernel itself alright.

(Refer Slide Time: 18:57)



So, we said that every process let us say this is P1 maps the kernel starting at you know starting a 2GB, let us say right and we said this mapping of the kernel is basically a one to one mapping to the physical address space. This is the physical address space, and this is going from 0 to M, then this is going to from 2 GB to 2 GB plus M right.

So, it is a completely contiguous mapping in the kernel right. That is what we saw that the kernel just maps the entire physical memory into a charger space; at least in xv 6 and we said other operating systems can recycle virtual address space for if you know the amount of physical memory is greater than what can be supported in the address space.

So, for this kind of a mapping which is completely contiguous it does not make sense to use small pages you can just use large pages right. So, you can just have you know 4 MB pages to store this mapping from for the kernel; that way, the kernel the size of the there few advantages of this number 1 – the size of the page table has reduced right because you only need few entries to map the kernel you have made the generality bigger.

So, the overhead that we had so, we said that every process needs to map the kernel. So, every process has to have this extra overhead of having these extra entries for the kernel address space. If you are using large pages, then this overhead has significantly decreased number 1. Number 2 – that most more important the TLB pressure has decreased.

The number of entries that are needed to be in the TLB to cache this mapping has increased and so, you can have better hit ratios in your TLB right. So, large pages can be used other places, but one place where they have a direct use, immediate use in the operating system designs that we are talking about you know you just use these large pages to map the kernel itself.

Student: Sir, while is saying while caching a page we (Refer Time: 21:04) to a 4MB in the cache it is very large value.

While catching a page, no, you mean in the TLB?

Student: Yes.

So, in the TLB you only store one mapping right.

Student: Yeah.

VPN to PPN. If you are using 4 kilobyte pages.

Student: We (Refer Time: 21:18) can get 2^{10} to the power 10.

You could potentially have 2^{10} entries for to map an entire 4 MB space right; on the other hand, if you have large pages then you will have only one entry for that entire space. But, large pages have problems of fragmentation and wastage of space potentially if you are not careful and so, that there is a trade off, but here is one here is one case where it is a no brainer to always use large pages. Yes, question?

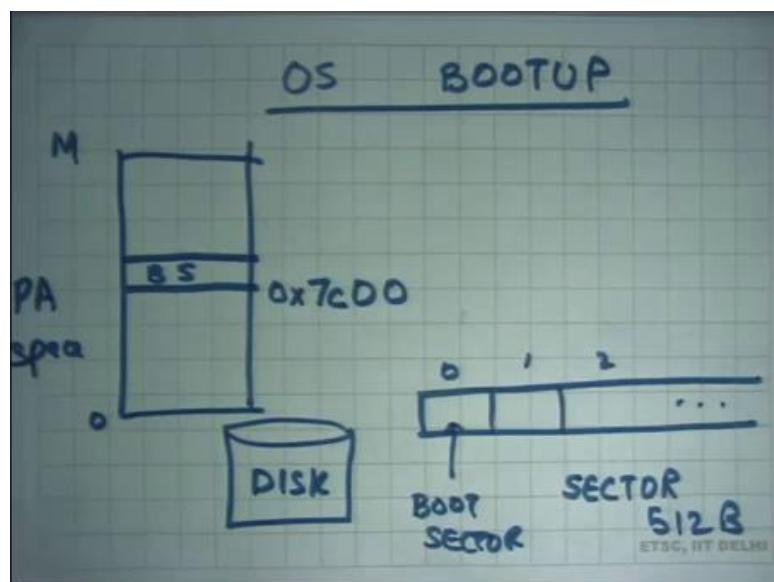
Student: Sir, since that space going to be constant so, why not use directly map bit? Why use any sort of paging mechanism?

So, question is why use any sort of paging mechanism to map this space. You know it will be great if the hardware could provide me a mechanism saying map the entire space

here, but the hardware does not do that right. I mean once you have enabled paging you have to go through the page table alright, and you know the one way the hardware designer code had said was you know have a other bit saying that this is not going to go through paging. This particular address is not going to go through paging you know it complex it makes the hardware even more complex.

So, you know one way to do that is just use the existing page table hardware and large pages it can be used both for the kernel and for other things also ok. So, the hardware designer does not necessarily want to complicate his hardware just to support the kernel space mapping when it he can achieve the same effect by using a more general mechanism of using large pages alright.

(Refer Slide Time: 22:58)



So, with that let us continue our discussion of from yesterday. We will be looking at how does the OS boots. Let us look at the OS bootup and we said let us say here is a disk and here are it is blocks block 0, 1, 2 and so on and we said this particular block or sector is special right.

So, each of these sectors is 512 bytes and sector number 0 are special, it is called the boot sector and the contract with the hardware is that it will load this boot sector at a particular memory location. It will copy this boot sector in the particular memory location.

So, let us say this is the physical address going from 0 to some value let us say M. So, it is going to copy these 512 bytes at some location the boot sector BS and the address at loads it is as it is as 6 7c00 alright. I mean these addresses are just historical in nature you know. This must have been the address at which it had loaded you know back in the 1980s and it still continuous because for backward compatibility we want that the operating system that was written in 1981 should still run right. So, for that reason it has to do the same thing ok.

So, and we said that what and so, what the. So, the operating system developer needs to write a boot sector code that should know where the kernel lives in the disk number 1, and it should load the kernel into memory and then jump to the kernel.

So, all that operation needs to be coded up in the boot sector and all this the code to do all this should fit inside 512 bytes I means that is the constraint and that is relatively easy to meet it is not a big deal and we are going to look at the boot sector code of xv6 alright ok. So, please take out your and this thing code listings.

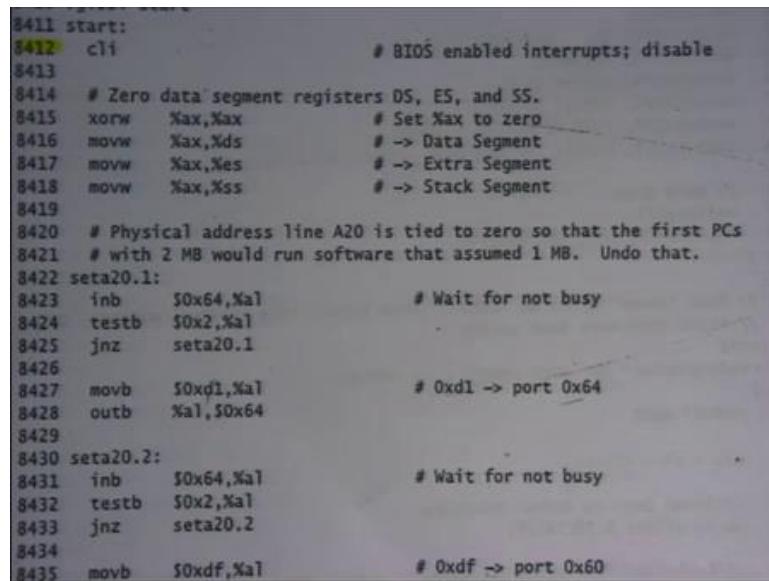
(Refer Slide Time: 25:22)

So, firstly, we are looking at code of xv 6 right this code will get compiled using let us say a compiler like GCC and we have discussed this before and then the code will get linked right. And, then you will get a an executable; that executable is what we call the kernel you know the xv6 links into what is called in an executable file which is called the

kernel right and that kernel will also and there will be another sort of file which will be the boot sector code right.

So, and the linker will set up the addresses and set up things in such a way that the boot sector code will live in the first sector of the disk number 1 and number 2 the boot sector code know as that will start at a certain address called 7c00 right that is the hardware specification.

(Refer Slide Time: 26:35)



The image shows a screenshot of assembly code from the xv6 operating system's boot sector. The code is annotated with line numbers (8411 through 8435) and comments. The assembly instructions include cli, xorw, movw, and testb, along with comments explaining the purpose of each instruction, such as setting registers to zero or disabling BIOS interrupts. The code is organized into sections like start, seta20.1, and seta20.2.

```
8411 start:  
8412     cli          # BIOS enabled interrupts; disable  
8413  
8414     # Zero data segment registers DS, ES, and SS.  
8415     xorw %ax,%ax    # Set %ax to zero  
8416     movw %ax,%ds    # -> Data Segment  
8417     movw %ax,%es    # -> Extra Segment  
8418     movw %ax,%ss    # -> Stack Segment  
8419  
8420     # Physical address line A20 is tied to zero so that the first PCs  
8421     # with 2 MB would run software that assumed 1 MB. Undo that.  
8422 seta20.1:  
8423     inb $0x64,%al      # Wait for not busy  
8424     testb $0x2,%al  
8425     jnz seta20.1  
8426  
8427     movb $0xd1,%al      # 0xd1 -> port 0x64  
8428     outb %al,$0x64  
8429  
8430 seta20.2:  
8431     inb $0x64,%al      # Wait for not busy  
8432     testb $0x2,%al  
8433     jnz seta20.2  
8434  
8435     movb $0xdf,%al      # 0xdf -> port 0x60
```

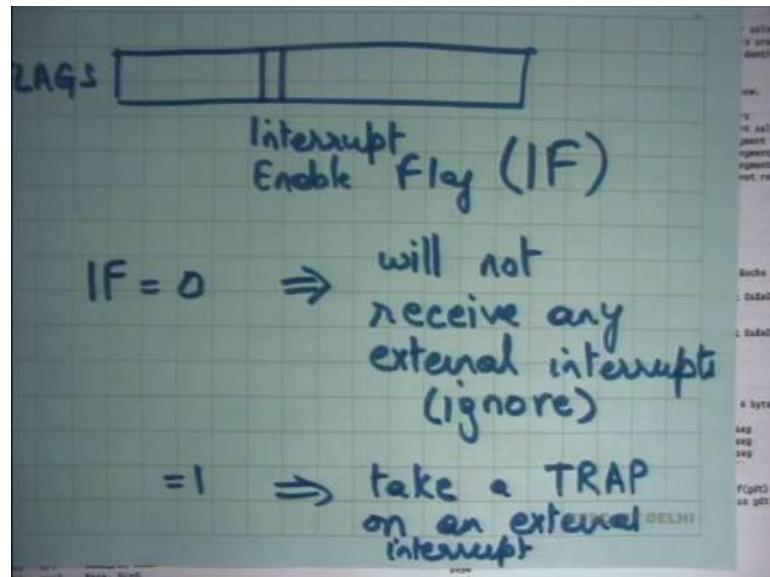
So, let us assume that the boot sector. So, the boot sector code actually starts at this point right 8412. So, this is the assembly code of the boot sector alright and the line 8409 is saying code 16. It basically means treat this code as a 16-bit code right. So, this is a 16-bit code.

The next line says global start which basically means consider this symbol start as a global symbol what it means we can talk about it later. So, this is the first instruction that gets executed on when the computer gets boot up booted up in xv6 the first instruction; in this case is cli; cli is basically just disabling interrupts right. So, what does the cli instruction do? Recall that the x86 architecture has this register called EFLAGS.

And, one of those one of the bits in the EFLAGS is whether interrupts are enabled or disabled. If interrupts are enabled, then an external device is allowed to assert the interrupt pin and my execution will switch to the handler immediately. If interrupts are

disabled even if the external devices assert interrupts pin, I will not switch to a handler right ok.

(Refer Slide Time: 28:00)



So, first recall that the x86 has a register called EFLAGS. One of the bits in the EFLAGS is what is called the interrupt flag or interrupt enable flag or IF ok. The semantics are if IF is equal to 0, implies will not receive any external interrupts. In other words, just ignore the external interrupt and if it is 1, then it is the usual thing you know take a trap on an external interrupt ok.

So, if IF is equal to 0, you just ignore the external interrupt; if I is equal to 1, then you will take a trap on the external interrupt and recall that the trap goes through the interrupt descriptor table to call the handler. Why the first instruction is clearing the interrupt?

Student: Because there is no handler.

Because I have not set up any handlers yet right. The BIOS may have had set up its own handlers some sort there is some you know BIOS that has run before the first instruction has run it may have set up its own handlers and, now I want to completely forget what the BIOS has done to the system. I want to now you know reinitialize the system according to myself and while I am doing that I just I do not want any disturbance from outside world right.

So, if an outside there is if there is a network packet coming or anything of that sort or you know disk wants me to want some attention just ignore all that. I am not in the state to actually be serve all these people. So, let us clear the interrupts alright. So, that is what. So, cli basically sets IF to 0 ok. Then the next thing you do is you say xor ax ax the effect of xoring register with itself is there you just 0 out the resistor ok. And, then you move ax to all the segment registers DS, ES, SS. Why am I doing this?

Student: (Refer Time: 30:31).

To have a flat address space right. I do not to have do not have a segmented address space. Let us set up my segment register recall that in 16-bit mode the segment registers the segmentation works by simply multiplying the segment register value by 16 and adding it to the.

Student: VA.

VA virtual address to get a physical address. So, I do not want any segmentation so, I just set it to 0. So, my virtual address is equal to physical address from now on. So, from now on if my virtual address is equal to physical address.

Student: Sir.

Yes?

Student: Like instead of xoring could we have directly moved dollar 0 into x?

Yes, could you have directly moved dollar 0 into x? Yes, you could have. Why is he doing it in this way? It turns out it is more efficient to do it and this is you know this is a standard thing that assembly programmers use that you know instead of moving 0 to ax just xor it with itself that turns out to be more efficient counter intuitively right. Anyways, I mean you could have done the other thing also.

Student: Sir, other than we could have xored all three of them with themselves?

So, you know, but see DS, ES, SS are special registers. They cannot only you know a certain instruction move can be used on them. You cannot use arithmetic instructions on segment registers. You can only use arithmetic instructions on general purpose registers which are a es a, b, c, d, esp, ebp, esi, edi alright. So, you can only use arithmetic and

you cannot even move an immediate value to the segment register directly, you can only move it through a general-purpose register. So, these are just constraints of the architecture alright. So, that is why he is doing it like this alright.

Then there is some code to allow addresses above 20 bits. So, original 8086 machines did not allow addresses which are greater than 20 bits right at that time the machine was 16-bit, but now we want a 32-bit architecture. So, there is some code to allow addresses above 20 bits in any case we can ignore this alright. So, let us ignore this. It is needed for a program to run, but it is not needed for us to understand what is going on alright. So, let us just ignore this.

(Refer Slide Time: 32:38)

```
8423  inb    $0x64,%al          # Wait for not busy
8424  testb   $0x2,%al
8425  jnz     seta20.1
8426
8427  movb    $0xd1,%al          # 0xd1 -> port 0x64
8428  outb    %al,$0x64
8429
8430 seta20.2:
8431  inb    $0x64,%al          # Wait for not busy
8432  testb   $0x2,%al
8433  jnz     seta20.2
8434
8435  movb    $0xdf,%al          # 0xdf -> port 0x60
8436  outb    %al,$0x60
8437
8438  # Switch from real to protected mode. Use a bootstrap GDT that makes
8439  # virtual addresses map directly to physical addresses so that the
8440  # effective memory map doesn't change during the transition.
8441 lgdt    gdtdesc
8442  movl    %cr0, %eax
8443  orl    SCRO_PE, %eax
8444  movl    %eax, %cr0
8445
8446
8447
```

And let us instead come to this instruction lgdt gdtdesc. What am I doing? I want to load up load a.

Student: Global descriptor.

Global descriptor table and notice that this lgdt instruction is actually a 32-bit instruction alright, it is not; it is not a 16 bit instruction because in 16 bit there was no gdt. In 16-bit segmentation was just multiplied a segment selector by something and that is it the gdt is only in this protected mode.

And, so what I am going to do in the next four lines is basically switch to 32 bit mode right, but before I switch to 32 bit mode I need to set up my gdt and all these things, so

that when I switch to it knows exactly where to where am I standing and all that right ok. So, what are the semantics of lgdt gdtdesc? Gdtdesc itself the descriptor is defined here at line 8487.

(Refer Slide Time: 33:33)

```
8463 # If bootmain returns (it shouldn't), trigger a Bochs
8470 # breakpoint if running under Bochs, then loop.
8471 movw $0x8a00, Xax           # 0x8a00 -> port 0x8a00
8472 movw Xax, Xdx
8473 outw Xax, Xdx
8474 movw $0x8ae0, Xax           # 0x8ae0 -> port 0x8a00
8475 outw Xax, Xdx
8476
8477 spin:
8478 jmp spin
8479
8480 # Bootstrap GDT
8481 .p2align 2                 # force 4 byte alignment
8482 gdt:
8483 SEG_NULLASM                # null seg
8484 SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
8485 SEG_ASM(STA_W, 0x0, 0xffffffff) # data seg
8486
8487 gdtdesc:
8488 .word (gdtdesc - gdt - 1)    # sizeof(gdt) - 1
8489 .long gdt                   # address gdt
8490
8491
8492
8493
8494
```

Student: (Refer Time: 33:34).

And, it contains the location of the gdt, and it contains the size of the gdt right. Recall that we said that gdt can specified by a size and a base also right. So, this is the base of the gdt; so, 0, 1, 2, 3 and there is a maximum size of the gdt. So, that is what you specify the gdtdesc and let us look at where gdt is that is at this line 8482 right which contains your segment descriptors right.

So, segment descriptors are nothing, but numbers right they are just numbers which basically specify this flag should be set up, this is the base, this is the limit and all that kind of stuff right. So, in this case here is a number called SEG NULLASM which basically says 0th descriptor is null – no never use it, nobody should ever use it alright.

The second one says it is using a macro. So, these are all macros. SEG NULLASM is the macro which will get macro expanded. So, you can actually browse the xv6 code to see exactly what number this SEG NULLASM represents, but you know whatever the number is it basically sets up the gdt 0th entry to say that this should never be accessed basically dereference. The second says SEG underscore ASM which is again a macro.

STA X which says give execute privileges. X is for execute STA R is saying give read privileges. So, these are basically specifying what flags I want in the gdt descriptor. 0 says what is the base of the gdt descriptor and this ffff says what is the limit of this gdt descriptor, alright. So, you use a macro to say construct the gdt descriptor with flags, execute and readable and base 0 and limit 2 to the power 32 minus 1 ok.

So, the intent is that the descriptor number 1 is going to be dereference for all your code right. So, you are going to load the value 1 into cs the descriptor number 1 into cs and so, code will always go through this and so, that is why it will become executable alright and all the other ones will have writable which means writable means it is both readable and writable and base is 0 and limit is again 2 to the power 32 minus 1 right.

So, you have a gdt of size 3, the 0th is a null entry, the first entry is an execute is pointing to an executable segment which is the entire address space, then the second entry is pointing to a writable segment which is again the entire address space. The developer has separated its code and data in this way. So, we will load code in cs and data in all the other segments right.

You may say could I have had a one a single segment which is readable, writable and executable and load that the same thing in everything that is also perhaps possible depending on you know whether x86 allows that kind of thing that the segment is both writable and executable alright.

But, in any case they are you know they are sharing the entire address space. It is not like cs is living somewhere else and all the other segments are living somewhere else. They actually living in the same space the different segment descriptors are only being used to check the type of access execute versus right clear alright ok.

So, see you will load the gdt and then you do some things to make sure that you are moving into 32-bit mode. So, CR0 PE basically says move into protected mode. So, the 32-bit modes that you have discussed is also called the protected mode because it offers protection right and so, when you and then you move this value into CR0. So, there is a control register 0 which indicates which mode I am running in and I am running in 16 bit mode, 32 bit mode or protected mode etcetera and so, that is how you move to 32 bit mode.

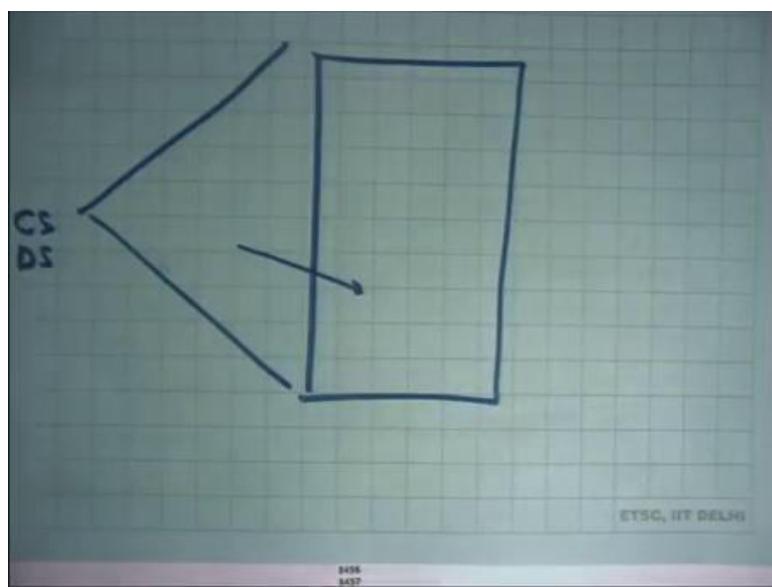
Student: Sir, why we have taken the limit for both data segment and code segment from 0 to 2 raised to the power 32 minus 1.

Why have we taken?

Student: The limit from 0 to 2 raised to power 32 minus (Refer Time: 37:50) because code and data are two different part work flow limits should be like 0 to for example m and from m to 2 raised to power 32 minus 1.

So, question is why both of them are referring to 0 you know are pointing to the same region 0 to 2 to the 32 minus 1? Why not code referring from 0 to some m and data from m to 2 to the power 32 minus 1? Well, I mean he could have done that, but the prefer I mean that complicates the programming model. You know sometimes you want to access code just like data.

(Refer Slide Time: 38:31)



Student: But we will not be differentiating over rather we want to execute it or we want to write over it.

Right. So, basically this kind of organization allows you to see a flat address space right. It does not matter which segment you are going through you will always VA will be always equal to PA right. On the other hand, if you do the other thing then I will have to worry about you know whether I am going to through this segment or that segment and if you have a flag address space it allows you to change code using directly it is address

and then execute it and because all of them are pointing to the same region you do not have to worry about you know CS is pointing here and DS is pointing here.

So, all these segments CS, DS etcetera they basically see identical things right. So, the program needs to only worry about the offset. It does not need to worry about the segment at all in this organization. So, it is a flat segmentation model right we have discussed this before.

(Refer Slide Time: 39:38)

```
8450 # Complete transition to 32-bit protected mode by using long jmp
8451 # to reload %cs and %esp. The segment descriptors are set up with no
8452 # translation, so that the mapping is still the identity mapping.
8453 ljmp $(SEG_KCODE<<3), $start32
8454      e=|
8455 .code32 # Tell assembler to generate 32-bit code now.
8456 start32:
8457 # Set up the protected-mode data segment registers
8458 movw $(SEG_KDATA<<3), %ax    # Our data segment selector
8459 movw %ax, %ds    # -> DS: Data Segment
8460 movw %ax, %es    # -> ES: Extra Segment
8461 movw %ax, %ss    # -> SS: Stack Segment
8462 movw $0, %ax    # Zero segments not ready for use
8463 movw %ax, %fs    # -> FS
8464 movw %ax, %gs    # -> GS
8465
8466 # Set up the stack pointer and call into C.
8467 movl $start, %esp
8468 call bootmain
8469
8470 # If bootmain returns (it shouldn't), trigger a Bochs
8471 # breakpoint if running under Bochs, then loop.
8472 movw $0x8a00, %ax    # 0x8a00 -> port 0x8a00
8473 movw %ax, %ds
```

So, I have moved into segmentation mode into 32-bit mode, but there is one more thing that the x86 architecture requires you to do which is to say that you know switch. So, right now what happened was I changed the control register 0 to say that I want to execute in 32 bit mode and then I executed this instruction called 1 jump right long jump we have seen this before, segment id and offset right.

Recall what the 11 jump does it just loads CS with this value the value of SEG KCODE is 1 alright. So, this is equal to 1. So, it just loads the first descriptor into CS; segment descriptor number 1 into CS and what is start 32? And, start 32 is just here alright a start 32 is the value or the address of this particular instruction whatever comes after start 32 alright.

So, it basically saying jump so, it is by using 1 jump it is basically causing CS to get overwritten; right now, CS was 0. Now, you are overwriting CS with 1; 1 shifted by 3

bits right. So, that way you basically are looking at the first segment descriptor and now you are actually executing through the gdt ok. And, start32 is the address of this. Once again because you know everything was 0, so, I did not have to worry about you know start 32 is an offset etcetera start 32 is just an address and offset is equal to address so no problem right.

So, now, at this point I have loaded the code segment and I have loaded the eip and I have moved into 32-bit mode. So, the moment I loaded the new code segment I basically have declared that I am executing in 32 bit mode and that is why this particular directive your code 32 is telling the hardware that telling the assembler that interpret all the next instructions the 32 bit instructions ok.

And, now the first few things that you do is that you load all the segment registers with k data k data is you know 2 let us say so, 2. So, you load segment number descriptor number 2 into ds, es, ss all the other segments right.

Student: Sir, why are the bits should be in the 3.

Recall that the segment selector. So, the segment register had the last 3 bits reserve for something else the top you know the segment selector itself was living only in the last in the top sort of 13 bits. So, that is why we are shifting it by 3 right. Also, recall that the last 2 bits of the CS register were meant to indicate the privilege level. In this case when I am shifting it by 3 the last 2 bits are.

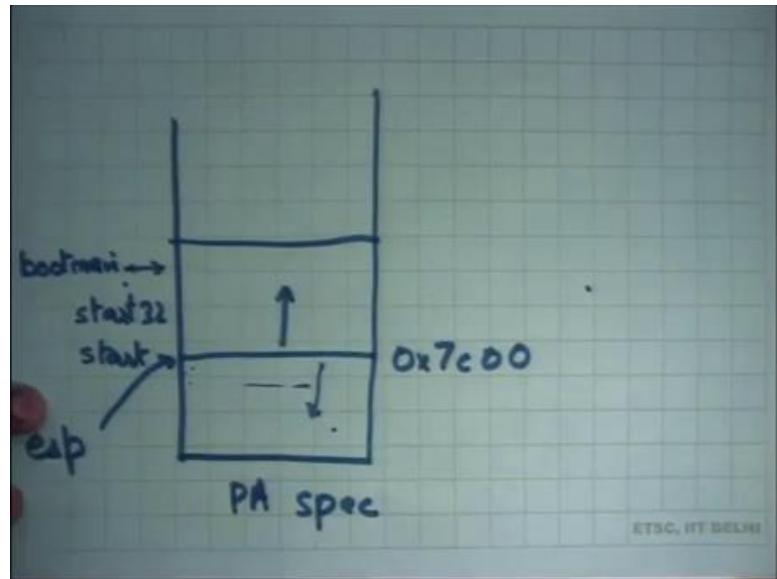
Student: 0.

0 and so, I am still executing in privilege level alright. So, I load up all the segment registers just like before, but this time is a different value SEG KDATA which is shifted left by 3 alright and then what I do is I move the dollar start to esp. What am I doing here? Let us see. What is dollar start? Dollar start was the address of this place which was.

Student: 0.

0x7c00 right? We said that you know that is where the code is going to start and so, the linker has organized it in such a way that the address of this place is 0x7c00 that is why we started at that place alright.

(Refer Slide Time: 43:35)



So, the value of dollar start is basically 0x7c00 and then putting it in esp. What I am really doing is I am initializing my stack alright. So, what happened was let us say this is my PA space and I said this is 0x7c00 all this code that we are looking at is actually living in this area right. So, this is start and let us say this is somewhere here to start 32 right and up to 512 bytes ok. So, all this is living in this area.

And, what I did right now was I set my esp register to point to this location and recall that the esp grows downwards. So, all these areas here can be used as stack now right. So, when I am going to make a function call the stack frame is going to get pushed somewhere here in the lower area ok. So, that is what I am doing here. When I say move, dollar start dollar percentage esp I am putting the value 7c00 into esp and then the next thing I do is I call a make a function call alright.

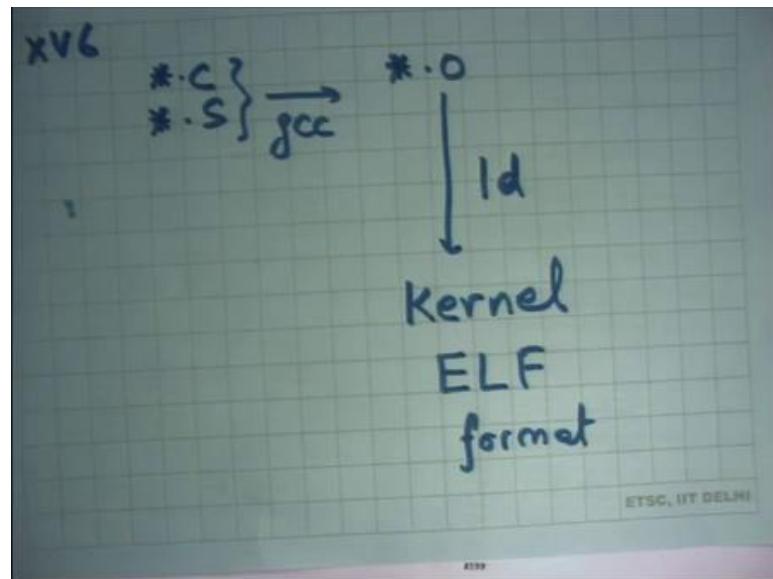
So, make a function call to boot main and so, the return address gets stored at 0x7c00 alright; actually, 0x7c00 minus 4 that is why I written address will get stored and esp will get decremented by 4 right ok. Where is boot main living? Where did this boot main come from? Boot main is so, we have at you know we have an we were executed the first few instruction is assembly. Programming in assembly is difficult. So, let us write the rest of our code in C alright.

So, we are going to. So, the boot sector itself the rest of the boot sector is living is has been written in C. So, boot main is also living in the boot sector somewhere. So,

somewhere here in this space itself there is boot main, alright and then linker appropriately arranged for it we are in that address and the code for boot main is in the next sheet 85 and now, we see some C code and that is you know more familiar and easier to sort of understand.

So, what this function is going to do is it is going to load the kernel from the disk into memory. So, you know 512 bytes is too small. So, I want to load the other bytes of the kernel into memory and jump to the first instruction of the kernel, that is what this function is going to do right and the compiled code of this function is living in the boot sector alright. And, so let us understand first how the kernel is organized.

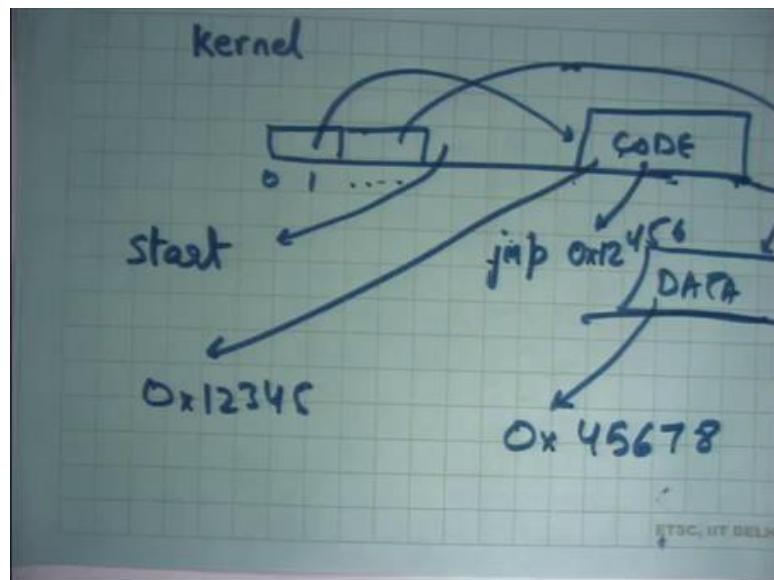
(Refer Slide Time: 46:53)



So, what happened was you had some dot C files and dot S files in your xv6 alright. You converted them to start dot O files and then you let us say g through gcc and then you linked all these dot O files and got some image which is called the kernel right. This kernel is an executable file in a format called ELF even your programs a dot O files etcetera they are also you know in this format called ELF.

And, this there is a there is standard format which is specification that you know this is the format of an executable file.

(Refer Slide Time: 47:49)



And, what the executable file will have is basically it will have things like if I look inside kernel it will say, by the way you know hey this is my file and you know this is byte number 0, byte number 1 and so on. And, you will say byte number 0 has to have certain header which will indicate where my code is. So, there will be some pointer here which is here is all your code right here is where it starts and here is where it ends so, all the size and all that.

And, then it will say you know let us say here is all your data in the file and so on right. It will also say that this code should be loaded at what address. So, it may say load code at address you know 12345 right and load data at address you know whatever 45678 right. So, it has all this information. It says, here is your code, here is the data, load code at this address, load data at this address alright. These are virtual addresses right I mean this is this basically whatever address space I am currently executing in.

So, if I am executing a program, I will look at the ELF file and the process has some address space and so, ELF file will tell me in this address. So, let us the process has an address. So, from 0 to 2 GB the ELF file will say load the code at address 5 MB and load the data at address 7 MB alright and that is it and here are the first instructions.

So, the other thing it has is start, where to start? So, you have loaded the code and you have loaded the data in the into the address space placed them there and now you want to know where I should start. So, what is the first instruction I should execute? In other

words, what should I initialize eib 2 right and so, that is also stored in this file basically says after you have loaded it set eib to this and you are good to go after that you just execute whatever you like and you will probably execute some instructions, you will make some system calls whatever you do right.

Similarly, in this case in the kernel case it is not really an it is not the kernel is not going to be loaded inside the process, but it is going to be loaded by the boot loader into physical address space initially right ok. Also, then you say that the code should be loaded at a certain address the code internally could have pointer to itself right.

So, example inside it I could say you know jump to some address 12456 right. So, 12456 should be meaningful right because I know that I am going to be loaded at 12345. So, 45 I know that what instruction is going to live at 456 and so, internally I could say you know jump to in 456 so, I know what instruction is going to get executed in that time. For example, you know functions are named by addresses, variables are named by addresses right.

So, a variable will live in the data section, a global variable will live in the data section right. A function will live in the code section and these things will have internally pointers to each other right. Code some instructions will have pointers to some functions right or some instructions will have point some data, but these are all at fixed addresses because the ELF has dictated that this particular data should be loaded at this address and you have already pre-computed the address of the variable inside the data section you require right.

So, similarly what is going to happen is that this kernel will say that this is my code and this is my data and we will say load this code at this address and load this data at this address and so on. And, what my boot loader is going to do is respect whatever the ELF file is telling in and put it at the right address. And, now and also the ELF file will contain the start address and based on that it is going to jump to the kernel right.

And, so the kernel has so, the boot sector job is over, now it is the kernel which takes over after that. So, the boot record loads the kernel and jump to its first instruction and how it does all this is basically dictated by the ELF file alright.

Let us stop here and we are going to discuss more code next time. So, highly recommend that you familiarize yourselves with xv6 and all that and we are going to do more of this example next class.

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

Lecture - 14
Loading the Kernel,
Initializing the Page table

Welcome to Operating Systems lecture 14. So, so far we have been looking at Xv6 code and we said an operating system writes some code to the boot sector which is the first 512 bytes of the disk, which is responsible for execute, starting execution in 16 bit mode and then loading the kernel from the disk and transferring control to it right. We saw that the first few instructions in the boot sector are written in assembly and very soon the assembly instructions initialize the stack and initialize the environment to a 32 bit environment, initialize the address space using segmentation and then jump to C code right.

So, the reason that it jumps to C code is because it is easier to program in C, it is a slightly higher level language and so, it you know it makes it easier to read the program as opposed to keep coding in assembly. On the other hand there are some things that can only be done in assembly right, because things like execution of the LGDT instruction right; there is no C counter part of the LJUMP instruction or there is no C counter part of the L jump instruction right. So, there is certain instructions that have to be done in the assembly and for other things you just use C right.

There was a question just now that why C why not some even higher-level language like Java, right. So, I mean it is a tradeoff, C is reasonably high level and yet the compilation of C usually gives you know the assembly code that comes out of compiling C is usually very close to in performance to hand written assembly code right. And so, performance is not lost really, significantly if you are writing programs in C; also, the environment that needs to be initialized for a C program is pretty small.

If you saw what got initialized, you initialize an address space and you initialize the stack right and that is it and you could jump to a C program. On the other hand initializing in the environment for something like the Java runtime which involves a JVM, it involves garbage collection and things like that will involve more steps to be

able to run execute the first Java instruction right. Also, Java, because the I mean languages like java assume the presence of a virtual machine right. So, the reason Java claims to be a platform independent language is because there is a virtual machine layer sitting JVM, Java Virtual Machine layer sitting between the real hardware and the program. And the so, the JVM needs to actually induces some level of overhead, performance overhead.

So, if the performance of an operating system is the lowest layer of software and performance is often critical right. So, because if your lowest layer of software is not performing to the best then every other thing will get affected. So, performance is really critical so, you know you make a tradeoff between convenience and performance and. It is actually so, as we as I said it is not necessary that java is more convenient either so, alright. So, we were looking at the boot sector code.

(Refer Slide Time: 03:53)

```
8411 start:
8412 cli                                # BIOS enabled interrupts; disable
8413
8414 # Zero data segment registers DS, ES, and SS.
8415 xorw %ax,%ax                         # Set %ax to zero
8416 movw %ax,%ds                          # -> Data Segment
8417 movw %ax,%es                          # -> Extra Segment
8418 movw %ax,%ss                          # -> Stack Segment
8419
8420 # Physical address line A20 is tied to zero so that the first PCs
8421 # with 2 MB would run software that assumed 1 MB. Undo that.
8422 seta20.1:
8423 inb $0x64,%al                        # Wait for not busy
8424 testb $0x2,%al
8425 jnz seta20.1
8426
8427 movb $0xd1,%al                        # 0xd1 -> port 0x64
8428 outb %al,$0x64
8429
8430 seta20.2:
8431 inb $0x64,%al                        # Wait for not busy
8432 testb $0x2,%al
8433 jnz seta20.2
8434
NPTEL
```

And we said that the first instruction that executes is a cli instruction cli, which clears the interrupts because I am I cannot just I have not installed any handler at this point. So, let us just clear the interrupts.

(Refer Slide Time: 04:09)

```
8425 jnz    seta20.1
8426
8427 movb   $0xd1,%al          # 0xd1 -> port 0x64
8428 outb   %al,$0x64
8429
8430 seta20.2:
8431 inb    $0x64,%al          # Wait for not busy
8432 testb  $0x2,%al
8433 jnz    seta20.2
8434
8435 movb   $0xdf,%al          # 0xdf -> port 0x60
8436 outb   %al,$0x60
8437
8438 # Switch from real to protected mode. Use a bootstrap GDT that ma
8439 # virtual addresses map directly to physical addresses so that the
8440 # effective memory map doesn't change during the transition.
8441 lgdt   gdtdesc
8442 movl   %cr0, %eax
8443 orl   SC0_PE, %eax
8444 movl   %eax, %cr0
8445
8446
8447 NPTEL
8448
8449
```

Then I do some things to enable 20-bit addressing, above 20 bit addressing; then I load the global descriptor table, we also saw that the global descriptor table had three segments in it, three descriptors.

(Refer Slide Time: 04:15)

```
8473 movw   %ax, %dx
8474 outw   %ax, %dx
8475 movw   $0x1ae0, %ax          # 0x8ae0 -> port 0x8a00
8476 outw   %ax, %dx
8477 spin:
8478 jmp    spin
8479
8480 # Bootstrap GDT
8481 .p2align 2                  # force 4 byte alignm
8482 gdt:
8483 SEG_NULLSEG                 # null seg
8484 SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
8485 SEG_ASM(STA_W, 0x0, 0xffffffff) # data seg
8486
8487 gdtdesc:
8488 .word  (gdtdesc - gdt - 1)      # sizeof(gdt) - 1
8489 .long  gdt                   # address gdt
8490
8491
8492
8493
8494
8495
8496
8497
```

The null descriptor, a descriptor which points to 0, to a segment from 0 till 2 to the power 32 minus 1 with execute and read privileges right, and then another segment once again from 0 to 2 to the power 32 minus 1, but with right privileges. And the intent is that I am going to use the segment descriptor number 1 for my code segment and I am

going to use the segment descriptor number 2 for the other segments right D S data segment, S S tax segment, E S the segment use those strings ok.

(Refer Slide Time: 04:55)

```
8450 # Complete transition to 32-bit protected mode by using long jmp
8451 # to reload %cs and %eip. The segment descriptors are set up with
8452 # translation, so that the mapping is still the identity mapping.
8453 ljmp $(SEG_KCODE<<3), $start32
8454     e=|
8455 .code32 # Tell assembler to generate 32-bit code now.
8456 start32:
8457     # Set up the protected-mode data segment registers
8458     movw $SEG_KDATA<<3, %ax    # Our data segment selector
8459     movw %ax, %ds    # -> DS: Data Segment
8460     movw %ax, %es    # -> ES: Extra Segment
8461     movw %ax, %ss    # -> SS: Stack Segment
8462     movw $0, %ax    # Zero segments not ready for use
8463     movw %ax, %fs    # -> FS
8464     movw %ax, %gs    # -> GS
8465
8466     # Set up the stack pointer and call into C.
8467     movl $start, %esp
8468     call bootmain
8469
8470     # If bootmain returns (it shouldn't), trigger a Bochs
8471     # NPTL breakpoint if running under Bochs, then loop.
8472     movw $0x8a00, %ax    # 0x8a00 -> port 0x8a00
```

And so, I initialize the GDT, I execute some instructions to set up a CR 0, control register 0 to switch to protected mode. And finally, I call an 1 jump instruction which loads the C S with SEG K code which is nothing but 1; which means you are loading the first SEG, the number 1 shifted by 3 bits on the left to CS. So, that CS now points 2 descriptor number 1 in the GDT, right. So, C S gets loaded with descriptor number 1, eip gets loaded with the address of start 32. And, start 32 is nothing but a function or a symbol that is just defined next so whatever is the address of this, that gets loaded in eip.

So, what you are going to do is, you are going to start executing this code in using the C S register and notice that because the CSS the segment which C S points 2 has base 0 you know start 32 can be used just like that; you know you do not have to worry about start 32 minus base or anything of that is how you know it is all start addressing. But what it also does is it tells the processor that from now on you are going to be executing in 32-bit mode right, and you are going to be executing in privileged mode. And so, all these instructions are compiled in 32 bit mode and that is what this directed dot code 32 is telling the assembler that, compile these instructions in 32 bit mode or assemble these instructions in 32 bit mode, right.

So, as you saw the same instruction has different representation in 16-bit mode and a different representation in 32-bit mode. So, these instructions have to be assembled in 32-bit mode and that is what this directive is telling the assembler, alright ok. And then what does they do, it basically look for the first thing it needs to do, before it makes any memory access is to initialize its segments right; because all memory accesses go through some segments register right. Recall that the default segment register for all memory accesses the data segment DS, the default segment register for all stack accesses through ESP and EBP is SS.

And also, there are some instructions called string instructions for which the default addresses, default segment is ES, alright. FS and GS are two other segments that are not the default segment for any other instruction, but you can explicitly specify in the instruction that I want to use FS, right. So, what the programmer is doing here is that; he is loading SEG K data, which is descriptor number 2 in privileged mode, last two bits are still 0 in ds, es and ss.

And then it is setting a x to 0 and then loading 0 into fs and gs. What is going on here? It is loading the null segment into fs and gs. So, the programmer does not intend to use fs and gs at all, in his program. So, there will be no instruction in my kernel which will ever use fs and gs, so it might just initialize it to the null segment. So, any access through the null, through these segment register will create a, generate a trap; but that should not really never happen because my kernel will never be executing through FS and GS alright ok.

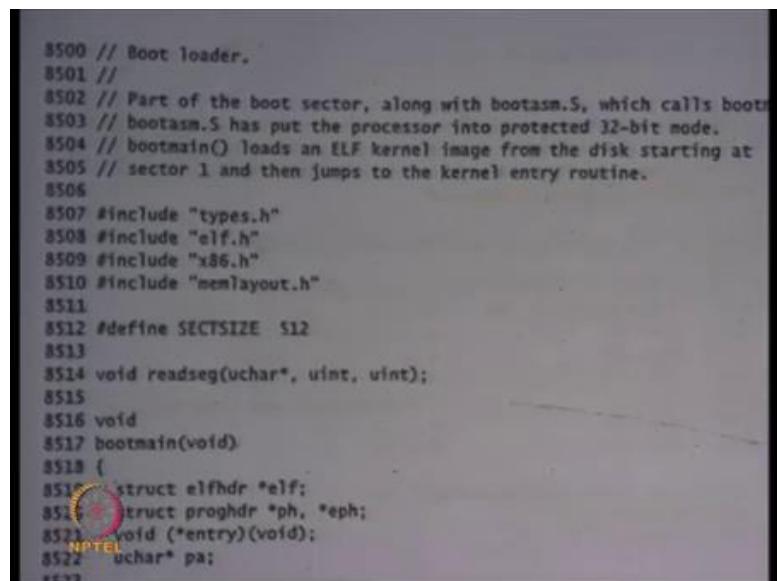
The next thing it does is it initializes the stack, we also saw how it initializes stack; it initializes a stack just below the code, just below where the code was loaded. So, it is code, the stack is going to grow downwards starting from 7C00. So, the all that space is basically junk because we are there is no code stored there and there is no other data structure that is stored there. So, it can be used for stack.

And right now, I just need the stack for calling my C function; the C function may have some local variables that will also get allocated on this stack, right. Recall that the C function will be compiled by GCC, as long as the caller of the C function conforms to the calling conventions which we have already seen, it is right. And so, here it is indeed conforming to the calling conventions; firstly, there are no arguments right,

secondly, I never expect this function to return, right. So, I do not actually. So, what boot main is going to do is, just going to assume that so, it has no arguments, so it is just going to start allocating on the stack and this function is never going to really return.

If it for any reason returns, which means it is a bug in your operating system, then there is some debugging code which allows you to see what happened ok. So, we can ignore that. So, now, let us look at the boot main function which is defined in C. And once again the assembly program is a caller of the C program right; and the C program is compiled using a specific compiler let us say GCC. And so, as long as the callers obeys the calling conventions it is right, and it is in this case ok.

(Refer Slide Time: 09:33)



```
8500 // Boot Loader.
8501 //
8502 // Part of the boot sector, along with bootasm.S, which calls bootmain()
8503 // bootasm.S has put the processor into protected 32-bit mode.
8504 // bootmain() loads an ELF kernel image from the disk starting at
8505 // sector 1 and then jumps to the kernel entry routine.
8506
8507 #include "types.h"
8508 #include "elf.h"
8509 #include "x86.h"
8510 #include "memlayout.h"
8511
8512 #define SECTSIZE 512
8513
8514 void readseg(uchar*, uint, uint);
8515
8516 void
8517 bootmain(void)
8518 {
8519     struct elfhdr *elf;
8520     struct proghdr *ph, *eph;
8521     void (*entry)(void);
8522     uchar* pa;
8523 }
```

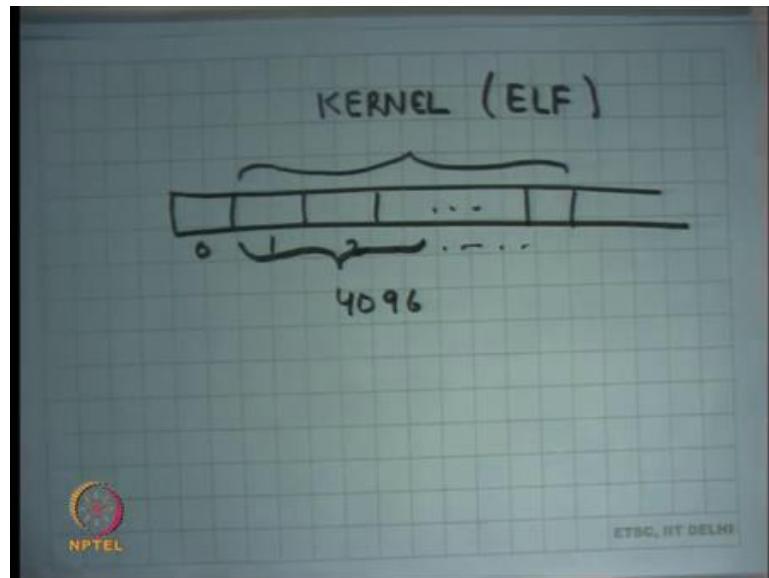
So, here is so, next sheet, sheet 85 is where you see the C code of the boot sector. Once again, the C code is living in the same 512 bytes of the boot sector, right ok. So, here is the function boot main, takes no arguments, returns nothing; actually, it never returns alright. And here are some local variables, they are going to be allocated on stack right, we understand that each stack is going to get decremented.

(Refer Slide Time: 10:03)

```
8518 {
8519   struct elfhdr *elf;
8520   struct proghdr *ph, *eph;
8521   void (*entry)(void);
8522   uchar* pa;
8523
8524   elf = (struct elfhdr*)0x10000; // scratch space
8525
8526   // Read 1st page off disk
8527   readseg((uchar*)elf, 4096, 0);
8528
8529   // Is this an ELF executable?
8530   if(elf->magic != ELF_MAGIC)
8531     return; // let bootasm.S handle error
8532
8533   // Load each program segment (ignores ph flags).
8534   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
8535   eph = ph + elf->phnum;
8536   for(; ph < eph; ph++){
8537     pa = (uchar*)ph->paddr;
8538     readseg(pa, ph->filesz, ph->off);
8539     if(ph->memsz > ph->filesz)
8540       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
8541
8542 }
```

The next thing it does is, it initializes some area at this address 10000, 10000 stands for 64 kb right. So, at 64 kilobytes memory address, it says you know I am going to use this as some scratch space to do some computation and so it initializes that area. And then it reads the first page of disk, right.

(Refer Slide Time: 10:41)



So, in this case, it is assuming that. So, let us say this is the disk logically speaking, and this was the sector number 0 which is the boot sector. What it is assuming is that; the kernel ELF is laid out starting at sector 1, alright. So, this is sector 1, 2 to some value.

Student: Sir.

All right and this is basically holding all the bytes of kernel in ELF format, alright. We also said that the elf format is an executable format which allows a loader to figure out where I should load these this the code, where I should load the data, what should be the first instruction should that should be executed and all that right. So, it is assuming that starting from this sector 1 and the first 4096 bytes.

Student: Contain.

Alright, contain what would be expected in the first few bytes of the ELF, the header, the header of the ELF file, right. So, that is what he is doing here. So, this particular command here is going to read 4096 bytes starting at sector number 1 into this area which is just some area that is not being used for any other purpose and type cast as an elf header. So, read those bytes and treat those bytes as the header of the elf file, alright. So, that is what this type casting means, alright. So, I read those bytes and then use and treat them as the ELF header. And then I when I treat type casted as an ELF header, then I check that you know some fields of this ELF header are correct. So, for example, there is a number called magic in the ELF header, with just checks whether it is indeed is an ELF file or not.

So that particular byte should be set to a certain value to indicate that it is indeed an ELF file. If that bit byte is not said to that particular value, then it is not an or it is a malformed ELF file or something. So, I you know loader should complete a border right there. So, that is what he is checking here, that the magic field is as I expected of the ELF header right, if not then.

Student: Sir, is not a creating a it has a character as a character array is let us been cast to character point.

Alright. So, it is reading 4096 characters that is why it is cast as a character array; but ELF itself has a type ELF header right. So, when you are going to. So, once you have. So, you are typecasting an ELF header to a character array and reading bytes as though they were characters; and now you are going to operate on this ELF variable as though it was a ELF header right, that is what you are doing.

So that is what, I mean the reason I can dereference dot magic is because the ELF header has a field called magic. So, I can just dereference it, right. And so, I check whether it is you know, whether it is correctly formed, it should be. And now the ELF header has a field called physical header offset. So, it basically says at what offset does the physical header lives, right.

And from that I get a pointer to the physical header or, so sorry, program header yeah not physical header the program header, right. So, the program header is obtained by adding this offset; offset is specified in the number as a number of bytes. So, you first typecast it as a pointer to bytes or a pointer to characters and then add this value; so, that many characters forward and then you typecast it back to a struct prog header.

So, you know typecast into a structure which represents a program header and the return value is basically the program header that you get, right. The program header itself is actually arranged as an array of program header. So, it is not, it is a program header is not a single program header, but basically in array of program headers you know so, a program header basically specifies the segment.

So, for example, here is the code segment, here is the data segment, you know here is the stack segment or whatever. And so, each segment is going to have some data, some values saying that no this is the offset at which this particular segment lives; this is the address at which this segment should be loaded right, and that is all and this is the size of the segment, alright

And so, the loader can look at that at program header and figured out this is the offset, and this is the address at which it needs to be loaded and this is the size. So, it just reads those many bytes from the disk or from the file and puts it in that address and that is what the job of the program header is, right. So, the program header has this information that where the segment should get loaded, where it lives in the file and what is a size.

And the loader is going to read this information to set it up, about based on what it says, right. So, that is what the boot loader is going to do, right. So, this is nothing but a loader except that it is a loader for the kernel right. In, when we saw when we talked about loading in the context of a process that was loading a process; so, the kernel loads a process, the boot loader loads a kernel, right.

So, that is a so, in this case and both of them and because operations are similar, you in the processes has use, process file uses the ELF format and the kernel file also uses the ELF format in this case right; and typically that is how it is done. So, you are just going to iterate over this array of program headers. So, till you get to you know so, there is another field which says how many program headers there are p h n u m. So, that is the end. So, this is end program header, ph say this is a start program header and you are going to just iterate till you reach the end program header and each time you are going to get the address at which it needs to be loaded; in this case it says p adder which is physical address.

It is going to read the segment off from the file at this offset, right. So, offset says what is the offset in the file from which this program, this segment needs to be loaded and it also knows how much how many bytes. So, file size says how many sizes. So, what is the size on file, what is the offset in the file and you are going to read that segment into this area called p a alright; and p a is given by p adder the physical address, right. So, we just read called read seg and I am going to read those one bytes into p a and p a is given by p adder so, you know basically doing what you, what a loader would do.

And finally, as an optimization ELF header allows you to specify the size on file and size in memory separately, right. So, the program header can have a different value for file size and different value for mem size. So, the idea is that a segment need not have all it is contains in the file, a segment may need to be you know let us say one megabytes large; but only 500 bytes of that one megabyte need to be initialized with some values, all other bytes can be initialized with 0, right.

So, here is that is what he is checking, if mem size is greater than file size for that particular segment then initialize the rest of the bytes with 0, right. So, s t o s b means store byte, starting from file size for these many bytes mem size minus file size, the byte 0. So, store 0 for all the bytes starting from file size till mem size, ok. So, that is what it is doing. And so, it so, it is basically executing the loader logic you know; similar loading loader logic will be inside the kernel to load an executable, here you are using the loader logic to load the kernel inside the boot loader, alright.

(Refer Slide Time: 19:13)

Kernel ELF format

\$ objdump -p Kernel

LOAD	vaddr 0x80100000	paddr 0x100000
align	filesz 0xb596	memsz 0x126fc

So, to understand this better let us look at. So, the kernel is an executable in ELF format right and there is there are some Unix utilities to be able to look inside an ELF file, alright. And so, one of the Unix utilities is o b j dump, alright. So, o b j dump and you can specify kernel as an argument, will print the contents of the ELF file in some human readable string format. So, you can actually you know till parse the file and print it and some human readable format, so that is what o b j dump does.

And so, let us look at what o b j dump of the kernel file? So, the kernel was compiled by a GCC and linked by some of the make files and all that; but let us look at the finished product and if you execute this command o b j dump dash p kernel, it basically prints out all the program headers in the ELF file, right. So, you can you know on your command from just do o b j dumped dash p kernel and it will print out the program headers.

And I am going to you know, I did this before the lecture and I noted down what were the contents of the kernel file, so that you know it will help our discussion. And so, here is what I saw, you know it said load which basically says it is a loadable segment it said v adder; so which said what is the virtual address at which this segment should be loaded.

And the value of this was 0x80 100000, right and then they were the p adder which whose value was 100000, okay. There was something called alignment, which says 2 to

the power 12 ok, it sets file size which said an hexadecimal b 5 9 6; I mean these values are not important, but mem size 0 x 1 2 6 f c, alright ok.

So, this is what this program segment was. So, this segment says that, you know it has. So, ELF allows you to have two different addresses; one word it calls v adder and other it calls p adder, and the loader is free to choose whichever it likes ok. And we want to see how it, which one is chosen when; it says at what should be the alignment. So, it says you know this particular program should be aligned at 2 to the power 12 boundary; it is a page size alignment.

The size on file is this value is the four-digit hexadecimal number and the size and memory is this value which is the five-digit hexadecimal number. So, what the loader is supposed to do is as load, supposed to load these many bytes from the read these many bytes on the disk and put them in memory and all the other bytes which is this minus this set them to 0, alright ok.

So here is an example of exactly what it does. Also notice that, carefully the developer has chosen v adder as or and p adder as some as values which are greater than equal to 1 and five zeros which is 1 MB right, why?

Student: Values

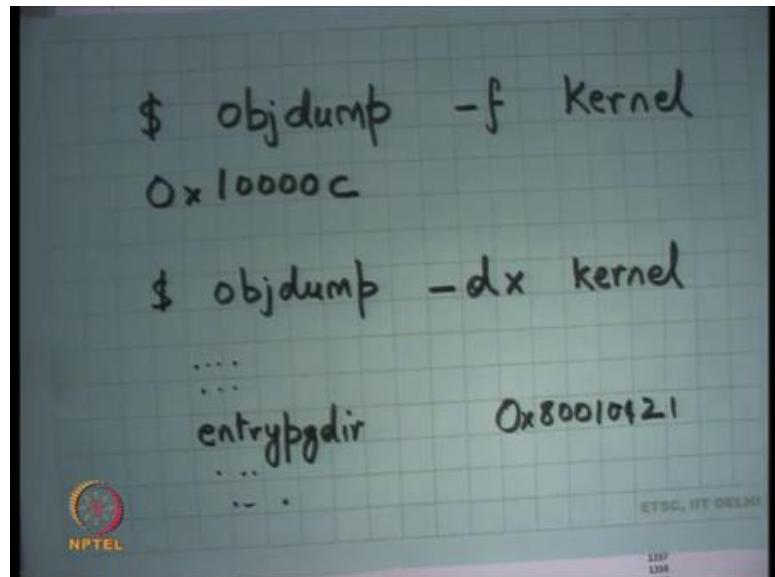
Why is this value not 0?

Because the first 1 MB or physical address space has memory map devices right, like the console and other things. So, there is some clutter there and you want to avoid over writing all that right; you do not want, you want real memory, you do not want a writing, you do not want things to be printed out on console. So, the real memory is definitely one, above 1 MB you will have real memory, right. So, the programmer chose to choose 1 MB in this case.

If he had chosen, could he have chosen a lesser value? Well he would have you to be very careful, if he is doing that so, that is not a great idea. Could he have to choose a greater value, for example, could he have chosen 2 MB? Yes, I mean it is completely valid to choose 2 MB; he is just choosing 1 MB. If you choose 2 MB or wasting more space in the bottom right, I mean why do you need to do that you just sort of keep it at 1

MB, alright ok. Also, if you want to look at the start address, you can use this command called objdump -f kernel, ok.

(Refer Slide Time: 24:31)



And what is going to you know. So, what I saw was basically this address 10000 and a C ok. So, this is basically saying where is my, what is the first instruction that you should execute. Notice that the first instruction is not necessarily the first byte of the program. If the first byte of the program is that 1 MB; the first, the starting point is 1 MB plus 12, alright 1000 C, alright.

So, with this understanding of how the kernel is, what the kernel has; let us go and look at the loader code.

(Refer Slide Time: 25:17)

```
8525 // Read 1st page off disk
8526 readseg((uchar*)elf, 4096, 0);
8527
8528 // Is this an ELF executable?
8529 if(elf->magic != ELF_MAGIC)
8530     return; // Let bootasm.S handle error
8531
8532 // Load each program segment (ignores ph flags).
8533 ph = (struct proghdr*)((uchar*)elf + elf->phoff);
8534 eph = ph + elf->phnum;
8535 for(; ph < eph; ph++){
8536     pa = (uchar*)ph->paddr;
8537     readseg(pa, ph->filesz, ph->off);
8538     if(ph->memsz > ph->filesz)
8539         stobs(pa + ph->filesz, 0, ph->memsz - ph->filesz);
8540     }
8541 }
8542 // Call the entry point from the ELF header.
8543 // Does not return
8544 entry = (void(*)(void))(elf->entry);
8545 entry();
8546
8547 }
```

So, here it is going to load the kernel from the disk to address 1 MB in physical address, right. So, notice that he is not using the v adder field of the program header; he is using the p adder field in the program header. The loader is free to choose whatever he likes right; but he is using the p adder field here, because the v adder is not even set up right now, right. 2 GB and above memory is not even mapped, it is only the bottom memory that is mapped; we are currently executing in physical address space, paging has not been enabled at, alright.

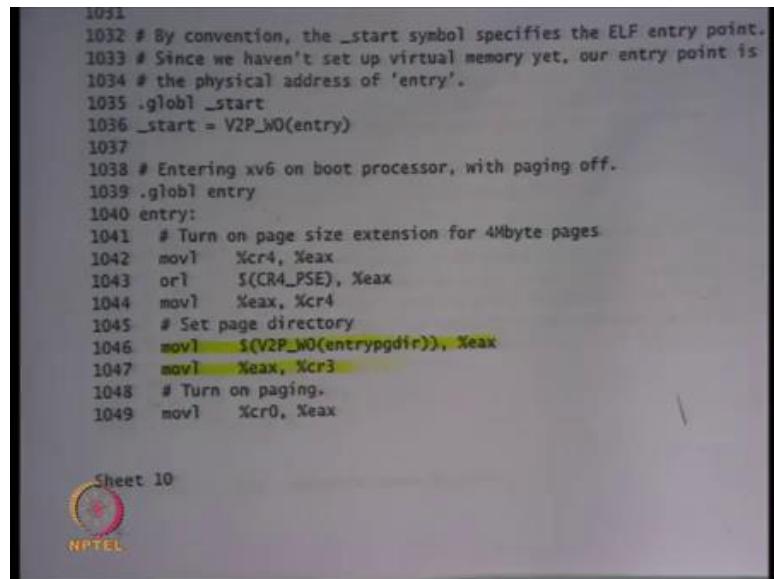
And so, firstly, xv6, looking at this you can be sure that xv6 will never be able to boot in any machine which has less than 1 MB space, right. Because you know, it is trying to dereference a location which is at 1 MB. In fact, it should have at least 1 MB plus whatever the mem size and the program header says; it does a minimum that xv6 needs, actually it needs a little more, alright. So, so it just. So, it is basically writing it to p adder because p adder make sense or it can just dereference p adder and it can write it there, right. So, the tool chain basically set up p adder such that, the boot loader will basically you read use the p adder values to load it the right place.

And finally, it just calls, it just figures a. So, the entry field in the ELF is basically that we are where the program should start execution, right and that is that 1000 C. And so, it basically reads that value from elf dot entry, once again it is in the header; and I just treat

it as a function, I typecast it to a function and I just call that function ok. So, at this point I am basically switching from the boot sector executable or to the kernel executable.

So, the kernel executable has been compiled to start at a 100 C and that is actually the code at entry dot s on sheet 10. So, let us look at sheet 10.

(Refer Slide Time: 27:33)



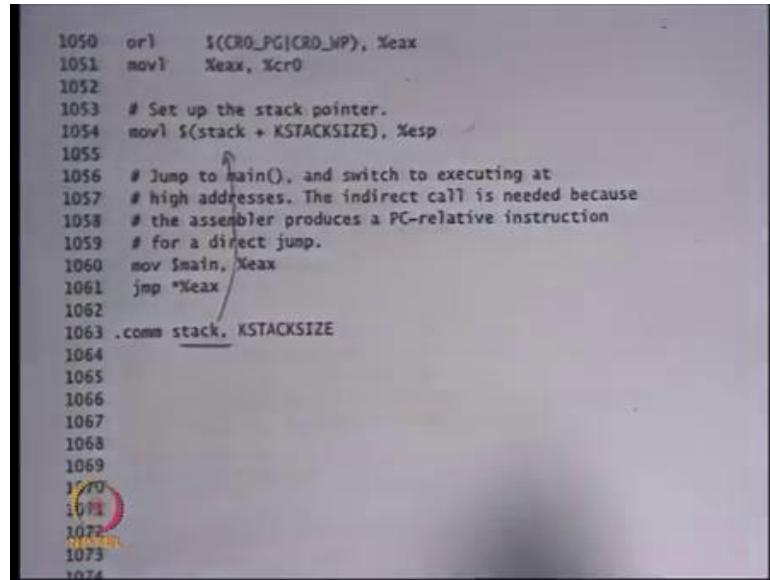
```
1031 # By convention, the _start symbol specifies the ELF entry point.
1032 # Since we haven't set up virtual memory yet, our entry point is
1033 # the physical address of 'entry'.
1034 .globl _start
1035 _start = V2P_W0(entry)
1036
1037 # Entering xv6 on boot processor, with paging off.
1038 .globl entry
1039 entry:
1040     # Turn on page size extension for 4Mbyte pages
1041     movl %cr4, %eax
1042     orl $CR4_PSE, %eax
1043     movl %eax, %cr4
1044     # Set page directory
1045     movl $(V2P_W0(entrypgdir)), %eax
1046     movl %eax, %cr3
1047     # Turn on paging.
1048     movl %cr0, %eax
1049
```

So, once again the first few instructions of the kernel are implemented in assembly and very soon it is going to jump into C code, right. So, now the kernel has forgotten about the boot sector completely, another kernel will initialize itself. The kernel assuming that there is the physical address space that is mapped, so segments have been set up; but the first thing the kernel is going to do is enable paging.

So, let us see how it enables paging; firstly, it executes some instructions to enable to allow large pages. So, recall that large pages help in you know, reducing the number of pages that you required. So, it first enables large pages, you know you can safely ignore these instructions just to know that they are there. And finally, here is the most important instruction it loads, it uses a macro called V 2 P, which basically converts a virtual address to a physical address. Entry page dir is just a, it is going to be compiled as, the value of entry page dir will be a virtual address in the compiled kernel. So, it will be some value above 2 GB, right.

V 2 P what it does is, it just subtracts 2 GB from whatever the value is, to get the physical address. Recall that the physical address is just a, the virtual address in the kernel minus 2 GB, right. So, V 2 P is just computing the physical address of entry page dir, moving it into this register e a x and then moving that register to c r 3, so loading a page directory into c r 3, alright.

(Refer Slide Time: 29:17)



```
1050 orl $(CRO_PG|CRO_WP), %eax
1051 movl %eax, %cr0
1052
1053 # Set up the stack pointer.
1054 movl $(stack + KSTACKSIZE), %esp
1055
1056 # Jump to main(), and switch to executing at
1057 # high addresses. The indirect call is needed because
1058 # the assembler produces a PC-relative instruction
1059 # for a direct jump.
1060 movl $main, %eax
1061 jmp *%eax
1062
1063 .comm stack, KSTACKSIZE
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
```

And the next thing it is going to do is, it is going to enable paging. So, move c r 0 e a x, you know some instructions like you know enable paging. So, it is setting some flags or them or-ing some flags and then moving it to c r 0, it basically enables paging, right.

So, once again these instructions are not important, but just know that these are enabling paging. So, let us understand exactly what the contents of the entry page dir are, right. So, that is going to so, as soon as paging is enabled your address space has changed, right. So, the same address now means something else potentially, right. So, let us understand what the entry page dir is and that is sheet 13.

(Refer Slide Time: 29:59)

```
1305
1306 // Boot page table used in entry.S and entryother.S.
1307 // Page directories (and page tables), must start on a page boundary,
1308 // hence the "__aligned__" attribute.
1309 // Use PTE_PS in page directory entry to enable 4Mbyte pages.
1310 __attribute__((__aligned__(PGSIZE)))
1311 pde_t entrypgdir[NPDENTRIES] = {
1312     // Map VA's [0, 4MB) to PA's [0, 4MB)
1313     [0] = (0) | PTE_P | PTE_W | PTE_PS,
1314     // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1315     [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1316 };
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328 NPTEL
1329
1330
```

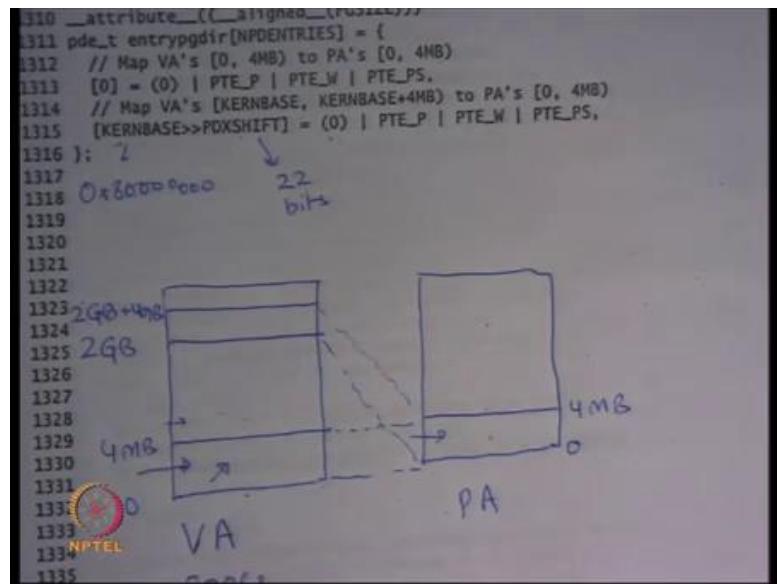
So, entry page dir is declared as a global variable in the kernel space, right. And so, that is fine right; it can entry page dir could have been a global variable, it could have been a heap variable whatever it does not matter. What matters it should have an address and it should have an address in the virtual address space of the kernel and what that instruction was doing was converting the virtual address into a physical address and loading into c r 3.

What is also important is that for the time that this page directory needs to be used, this address stays as it is right; nobody is overwriting this address or writing some other things on this address. So, global variable serves as a nice place to, I mean global variable will have that space allocated to it for the lifetime of the execution. So, that is in order reasonably good place to store this thing. Entry page dir is an array of size NPD entries; NPD entries should be how many, how?

Student: 2 to the power 10.

2 to the power 10, right so, 2 to the power 10 entries in the page directory. The first entry that is 0th entry is pointing to physical address 0, these are the top twenty bits of physical address with the flags present, writeable and page size. Page size means, it is of large page, it is a 4 MB page right.

(Refer Slide Time: 31:23)



So, what this is doing is let us say. So, this, so let us say this is, let us say this is my VA space and let us say this is my PA space then 0 to 4 MB is being mapped to physical address 0, alright. So it has an identity mapping here, 0 to 4 MB; that is what the first entry is doing, it is mapping 0 to 4 MB in the virtual address space to 0 to 4 MB in the physical address space, right. The 0th entry is also pointing to a physical address 0, right. And the 0th entry of the PTE is mapping 0 to 4 MB, right for large pages, clear.

The next thing it does it says, look at this entry KERN base shifted by PDX shift. So, what is KERN base? Kern base is 2 GB. So, that is the base of the kernel one two three four five six seven, so that is 2 GB right. And PDX shift is 22 bits, right, so you shift and address 22 bits to get the PDX number, the page directory number, so that is 22 bits, right. So, it is basically saying, look at the look at address number 2 GB, look at the address 2 GB in the virtual address space. And so, this area let us say this was 2 GB to so, it just one entry and this entry also has present writeable and page size bit set. So, it is 2 GB to 2 GB plus.

Student: 4 MB

4 MB, right and where is it mapping it? It is mapping it to 0, once again. So, physical address is 0, right. I hope you are being able to read this right, it is basically saying this is the value of this particular entry. The value is basically, the top 20 bits of this value

integrate the physical address, and these indicate the flags, right. So, in this case the physical address has again 0. So, what is it doing? It is mapping this.

Student: At the same place.

At the same place, ok. So, that is how it has set up the virtual address space. So, the moment it enabled paging, the virtual address space has changed. Before it enabled paging; if you for example, said you know let us say you opened this in GDB, and you said show me the contents of this address. And after you enable paging you said show me the content of this address, you would get the same value right; because then there is an identity mapping, right.

So, the paging did not change the address range, the contents of the address range from 0 to 4 MB right; but it did change the contents of addresses which are above 4 MB, right. Earlier when there was no paging, I could have accessed the address 8 MB; now when I have paging, I cannot access access the address 8 MB, right. Because the 8 MB is not mapped right; but 4 MB is mapped, and it is mapped identically. So, 0 to 4 MB is identical, as it was before and after.

Similarly, earlier if I had said I want to access address 2 GB plus 10 let us say, it would have given me I cannot access that address, was the physical memory is not that large. But now if I say access address 2 GB plus 10, then I will get an answer and the answer will be the value of the byte at physical memory address 10, ok. So, the moment you turn on paging the address space has changed; but the programmer is clever, he ensures that right now all the code and data and stack are living in 0 to 4 MB.

And so, as soon as you turn on paging, these still remain, right. So, it is not like the carpet has been pulled under my feet. The addresses that I am using currently are still valid, as soon as I turn on paging and that is the reason, I basically have two mappings from a kernel and user.

What the kernel is going to do next is? Recall that the address space layout of processes that the kernel uses 2 GB and above, and the process uses, the user space uses 0 to 2 GB. So, ideally this mapping should not be there, right. So, this page table is only a temporary page table alright, just to initialize. So, because and you needed the temporary page table to make sure that as soon as you turned on paging, you know it is not like I

cannot execute anymore, right; because all your stack and code pointers are still in this area.

So, what the kernels going to do is; first initialize this, then he is going to jump from here to here and once he is in this area, then he is going to remove this mapping and now he can use that for processes, ok. So, there is a temporary page table which has both the areas mapped, 0 and 2 GB. And during that it just switches from physical addresses to virtual addresses and now it can use the lower addresses for use the processes, alright it is clear yeah question.

Student: Sir why do not we doing that V 2 P in that in the GB space?

Why are we doing V 2 P of entry page dir?

Student: Because so, there was no entry page directory system.

So, why are we doing V 2 P of entry page dir?

(Refer Slide Time: 37:21)

```
0911 };
0912
0913 // Set up a normal interrupt/trap gate descriptor.
0914 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0915 // - interrupt gate clears FI_IF, trap gate leaves FI_IF alone
0916 // - sel: Code segment selector for interrupt/trap handler
0917 // - off: Offset in code segment for interrupt/trap handler
0918 // - dpl: Descriptor Privilege Level. This symbol specifies the LTR entry point.
1033 # Since we haven't set up virtual memory yet, our entry point is
1034 # the physical address of 'entry'.
1035 .globl _start
1036 _start = V2P_W0(entry)
1037
1038 # Entering xv6 on boot processor, with paging off.
1039 .globl entry
1040 entry:
1041 # Turn on page size extension for 4Mbyte pages
1042 movl %cr4, %eax
1043 orl $CR4_PSE, %eax
1044 movl %eax, %cr4
1045 # Set page directory
1046 movl $(V2P_W0(entrypgdir)), %eax
1047 movl %eax, %cr3
1048 # Turn on paging.
1049 movl %cr0, %eax
```

Student: Yeah.

Can somebody answer that? What is the value of entry page dir?

So, the kernel has been compiled such that all the addresses are in 2 GB are in the range 2 GB and above, alright. So, the kernel itself has all the addresses in 2 GB and above.

And so, entry, the value of entry page dir will also be a value that is 2 GB and above, that is a virtual address. So, you need to convert a virtual address to a physical address before you loaded into the c r 3 register, right. So, entry page dir value, entry page dir is at a global variable, right. And what is the address of a global variable? It is a virtual address and it has been compiled to have an address which is 2 GB and above, right.

Because most of the time the kernel will run or actually almost all the time the kernel will run in virtual address space, which is 8 GB and above. So, the kernel has been compiled to assume that it lives in 8 GB and above address space, the virtual address space right. And so, all these symbols in the kernel have virtual addresses, which will be 2 GB and above. What you need to load into c r 3 is a physical address; recall that c r 3 takes only physical address, right. So, I need to convert the virtual address into a physical address and that just means subtracting it by 2 GB, because kernel has a one to one mapping and then loading into c r 3.

Student: Sir, but if paging was not enabled how are we having the virtual addresses earlier?

If paging is not enabled how are we having virtual addresses earlier? See we do not have, I mean the kernel has been compiled to assume that it will run in a virtual address space, at and it will start at address 2 GB and above.

Right. So, that is and that is the reason entry page dir has a value. So, that is a compiled value, right that is a compiled time generated value. At compilation time you said entry page dir will have a certain value and those values have to be 2 GB and above. Because entire kernel all the code and everything all those instructions have an address 2 GB and above. So, you know just to make this clear, if you ever do let us say o b j dump dash d x let us say right, kernel. You will be able to see all the symbols, alright. And one of those symbols will be entry page dir and its value will be something you know, which is above 2 GB 8 0 0 let us say 1 0 1 2 1 ok.

So, the kernel has been compiled to believe that it is going to be executing in the virtual address space, right. And so that is why. So, when you say entry page dir you are going to get a virtual address and you need to convert into a physical address before you actually load into c r 3, ok.

Student: Sir after the paging has been enabled then we will not we have to use V 2 P

After the paging has been enabled, we will not we need to use V 2 P.

How many say yes?

How many say no? Ok, 6 or 7; you still need to use V 2 P; because c r 3 has to have a physical address, right. C r 3 needs to have a physical address. So, if you are loading c r 3, only a physical address can be loaded into c r 3; you still need to have V 2 P, ok. So, I have initialized paging and once again what I am going to do is, I am going to initialize my stack, just like the boot sector initialize the stack before jumping into C codes. I want to jump into C code, and I am going to initialize the stack; the stack is declared as this variable of size kstack size, right. So, we allocate some value of stack, it is 4 kilobytes and you move the top of that value into e s p, so that the stack is grouped downwards.

So, you move into stack e s p and now you jump into the main function of the kernel which is, will be written in C, right. The way you jump into the main function of kernel in this cases, you move main into e a x and you jump you dereference that using star e a x. Notice that he is using an indirect jump to jump to the main kernel, you could have, could you have used a direct jump like jump dollar main straightaway. In general, you can use direct jump, but in this case you cannot and why etcetera let us just defer that discussion for later. Let us just say that it is required that you use an indirect jump to do this and I am going to discuss why you cannot use the direct jump here.

Student: Why are there stars before e a x?

Why is this star before e a x? Basically, saying that dereference e a x and whatever value you get set e i p to that, that is what the semantics are. So, so this is just a syntax AT&T syntax of the x86 instructions; it just jump star to some address, alright. So, what happened here in the assembly code of the kernel, it just enabled paging, it enabled paging such that, the address is it is currently running in are still valid; but new addresses have become valid 2 GB and above and now it is jumping to the kernel, the main the C code.

The main, the variable main itself is a variable that is compiled using the C compiler and the value of the main variable itself will be a virtual address, ok. Just like entry page they

had a virtual address which means 2 GB and above; the variable main will also have a virtual address which is 2 GB and above. So, what am I actually doing here is that, from right now and what was the value of e i p right now? E i p was somewhere between 0 and 4 MB right now, right; recall that we never jumped anywhere between to 2 GB, right. We even the start value that we had in the kernel was 10000C.

So, as I just jumped to the physical address 1000C. Now I have initialized a virtual address and here is the first time I am going to jump to the virtual address, right. The variable main contains a virtual address which is above 2 GB, just like all other variables in the kernel. And so, here is where I am jumping from physical to virtual addresses; and here is where I can explain why you know you need an indirect branch. A direct branch a uses P C relative addressing, alright; so it basically says, if the target is x and I am at y, then the bytes that get actually written on those instruction are y minus x, right.

And so, it works if you are working in the same address space; but if you want to switch address spaces, that PC relative addressing does not work, ok. So, that is a short answer, you know if you are interested you can read more about it. But an indirect branch is basically you know, it is basically setting up e a x to that value, whatever 8 is you know 8 0 0 something and then saying jump star e a x; which means nothing but just replace the current e i p with this value, which is some address 2 GB and above.

And because that address has been mapped in the address table, I will be standing on solid ground; no, I will not fall. Also notice that I the page table only mapped 0 to 4 MB, which basically means, that the address of main should be less than 4 MB, right. And that is actually true, because if you look at this sheet that I had, the entire kernel size is 1 2 6 5 f c which is a 5 digit hexadecimal number and 4 MB is you know 6 digit or 7 digit number. So, it is ok, right.

So, the kernel itself has is you know, the developer is aware of the fact that the kernel is less than 4 MB and that is why it has only setup the first and the only one entry in the page directory, in the temporary page directory entry, right. If it knew that the kernel is bigger, it would have probably had to initialize more entries in the page directory, right ok. So, I have initialized the stack; what is the value of stack? Is it a virtual address or a physical address?

Student: Physical.

Student: Virtual.

Virtual

Student: Virtual.

The entire kernel has been complied with virtual addresses. So, all the variables in the kernel have only have virtual addresses, alright. So, any time you have you refer to a pointer it is virtual address. So, e s p itself it is now having a virtual address; the moment you call jump star e a x, e i p also has a virtual address. So, now, you will be executing completely in virtual address space ok. And now you are going to start initializing your data structures and your page table, so that you can now run other user processes, alright.

So, I am not going to go into the. So, it is enough for today, but let us just look at the main function very quickly. So, here is the main function.

(Refer Slide Time: 47:37)

```
1216 int main(void)
1217 {
1218     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1219     kmalloc(); // kernel page table
1220     mpinit(); // collect info about this machine
1221     Tapicinit();
1222     seginit(); // set up segments
1223     sprintf("\ncpu%d: starting xv6\n\n", cpu->id);
1224     picinit(); // interrupt controller
1225     ioapicinit(); // another interrupt controller
1226     consoleinit(); // I/O devices & their interrupts
1227     uartinit(); // serial port
1228     pinit(); // process table
1229     tvinit(); // trap vectors
1230     binit(); // buffer cache
1231     fileinit(); // file table
1232     finit(); // inode cache
1233     ideinit(); // disk
1234     if(!ismp)
1235         timerinit(); // uniprocessor timer
1236     startothers(); // start other processors
1237     kinit2(P2V(4*1024*1024), P2V(PHYSSTOP)); // must come after startothers()
1238     userinit(); // first user process
1239     // Finish setting up this processor in mpmain.
1240     mpmain();
1241     NPTL
1242 }
```

So, this is what gets called from the assembly code. So, stack has been set up, I just jump through the main function, alright. And what the main function does? It makes a lot of other function calls to initialize a lot of things, right. So, for example, it is initializing the physical page allocator here, initializing the kernel page table, it is a multiprocessor machine, it is a multiprocessor operating system so, it also initializes other operate, other processor in the system.

So, the way it works is only one processor boots up and then it initializes all the other processes. So, all that is done here, it initializes the IO devices, interrupt subsystems; at some point in this initialization it will reenable interrupts. Recall that so far, we have been running with interrupts disabled, cli was executed right in the beginning. So, at some point in this initialization, it will reenable interrupts after it has set up the interrupt handlers and finally, it will call you know, it will call user in it which will create the first user process.

And that user process will probably you know call fork or something to create more user processes, which will eventually call the shell for example. And now you can type command on the shell and potentially fork even more user processes and so on. So, that is how the user process going to get created. And you know and then you just let all the processes run simultaneously, right. So, multiprocessor; so, all processes can now execute, and they can see what the processes are, that need to be run and run them run concurrently, ok. But tomorrow I am going or the next lecture I am going to look at k v malloc in detail, alright.

So, we have seen that the kernel initialized a temporary page table to switch from physical to virtual address space. Now I am completely in virtual address space, so I can remove the mapping from the physical address space and so, that is what k v malloc is going to do. And now it will arrange it in such a way that each process has a separate page table and you can actually map user pages into that address space, ok.

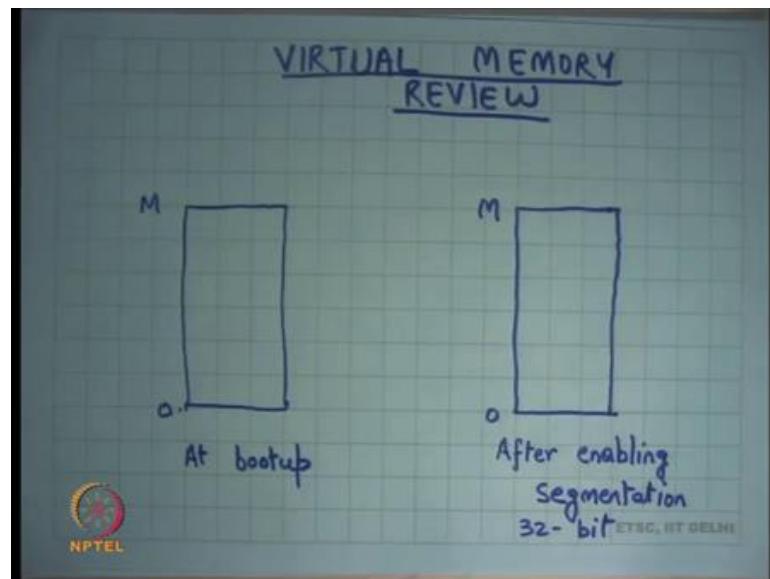
Thank you.

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

Lecture – 15
Setting up page tables for user processes

Welcome to Operating Systems lecture-15.

(Refer Slide Time: 00:30)



So, let us review the virtual memory subsystem as we have been looking at it. So, when you bootup the machine, we said that we start in the physical address space right. And so, if you use address x , then if x is less than M , then you had the physical memory at that address x right. You straight away hit the physical memory there is no translation involved. If you try to hit a physical memory address that is greater than the available memory let us say capital M is the amount of physical memory available in your system, then it will throw an error so that is not valid right.

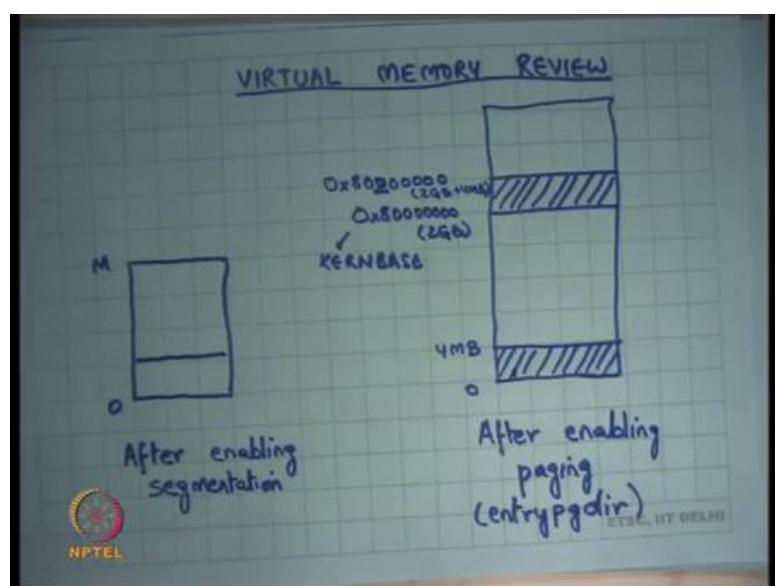
So, at bootup you have this simple system where there is no virtual memory subsystem and it is completely flat from 0 to M . Of course, recall that boot in 16-bit mode, so segmentation works in its primitive form, where you can add the segment register after multiplying it by 16 and all that. But for all practical purposes you know in kernel that we are studying sets all segment registers to 0, so you have a flat address space on 0 to M .

Then as soon as you enable segmentation in 32-bit mode, so you switch to you enable segmentation you initialize a global descriptor table and you switch 32-bit mode. The moment you do that your addresses are now getting translated through the segment hardware right. Each instruction has a default segment right. So, for example, if you are just making an access through some register, then it will be the default segment is ds right. The instruction itself will be the instruction pointer itself always go through cs and all that.

So, in theory you could enable segmentation and you could use segmentation to segment your address space. But in the kernel that we are studying it just sets base and limit to 0, base to 0 and limit to 2 to the power 32 minus 1. So, once again I have a completely flat address space right. So, if you would say I want to access address x, so let us say you fill in the value x into your eip register, it just goes into the physical memory at address x.

And as long as it is between 0 and M, you will see your valid byte; if it is greater than M, you will see an error right some kind of an exception will be thrown by the hardware. Similarly, ESP or any direct addressing, so these addresses can be generated in many ways through a register, through a direct addressing, through displaced addressing right all those things we have studied already right. In any case they all go through segmentation, and we saw that after enabling segmentation our address space is still the same 0 to M ok.

(Refer Slide Time: 03:12)



So, here is another figure. So, after enabling segmentation I had 0 to M, now the boot sector operates in this mode where only segmentation is enabled in 32-bit mode. And in that mode the boot sector, the boot sector was leaving in the first 512 bytes of the disk. So, the boot sector has code to load the kernel. The kernel lives from then sector number 1 till some value, and whatever that value is known to the boot sector the size of the kernel is known to the boot sector. And, so the boot sector loads that many sectors from the disk and puts them in memory.

While it is doing that, the address space is still 0 to M right. So, it picks up that kernel and sticks it somewhere here in this area right. We saw last time that it sticks the kernel at starting at address 1 MB right. So, it starts the it starts pasting the kernel starting at 1 MB and from there on it paste the kernel. And, recall that the size of the kernel was not very big either the size of the kernel was if I remember correctly definitely less than 1 MB right. So, you start you start the kernel from one M, you pick the kernel from that disk stick it at physical address 1 MB, and it will you know at most go till 2 MB right, all the kernel code and the kernel data.

Also recall that the kernel itself has been compiled with virtual addresses. So, all the symbols inside the kernel image have virtual addresses associated with them. So, if I say I want to branch to main, the address of main will be a virtual address right. It will be a virtual address in the sense that will be an address above this value 8 and seven 0s which is also called kern base. In our code, this called the kernel base right. So, all the symbols in our image have addresses above kern base, base or an above 2 GB.

So, all the symbols in that have addresses above 2 GB. So, anytime I dereference a symbol within my kernel code I am going to be trying to access a region above 2 GB. But in this address space, it is not possible right because assuming M is less than 2 GB, if I try to dereference a symbol in the kernel image, I will see an error, I will see an exception right, so not possible at this point.

So, the boot sector, so what the boot sector does it loads the kernel and it jumps to the first instruction of the kernel, and recall at the kernel was stored in elf format so the boot sector knows exactly where to start, and so the kernel the first instruction of the kernel get started. And the first instruction of the kernel leaves in this file called entry dot s

right. Now, the first few instructions of the kernel entry dot s are going to execute in this address space.

So, these instructions have to be very careful, they should not be dereferencing any kernel symbol, because all kernel symbols have addresses above 2 GB right. The moment it dereferences the kernel symbol, I will get an exception, not, it is not legal in this address space right. So, the first thing it does the entry dot s file does is it changes this address space by enabling paging right. So, it enables paging and it uses a page directory called entry page directory right.

And we saw last time that this entry page directory implements an address space where the first 4 MB are mapped identically. So, if you access an address x within 0 to 4 MB, you will hit physical memory at address x right. On the other hand, it also maps this address kern base to kern base plus 4 MB to the same location 0 to 4 MB right. So, if you access address kern base plus x, and assuming x is less than 4 MB, then you will hit physical address x right, so that is how the entry page dir was configured.

And, so the entry dot s code just switches from this address space to this address space as soon as it enables paging and checks sub CR3 point to entry page dir right. So, it points CR3 point to entry page dir enables paging and suddenly I am running in this address space. Recall that because the kernel itself all the instructions the entry dot s file itself was living in you know less than 4 MB space, so we set from 1 MB to 2 MB let say all the addresses you know the eip and ESP still remain valid, because the addresses from here are identical here 0 to 4 MB right. So, those remain valid.

You do not know it is not like you know I am standing on the ground and the ground has been taken away from it is not true because this as long as the kernel is less than 4 MB, the kernel safely mapped right, so the next instruction can execute alright. So, you enable paging, but ensure you ensured that the ground that I am standing on still stays as soon as I switch the address space right ok. But in this new table, there is an extra mapping and this extra this new this table is going to be used to switch from here to here right.

Recall that the kernel all the symbols in the kernel have values which are in this range. So, I should not be dereferencing any symbol from here. The only thing I am going to do is I am going to look, so in this there are some symbols like there is the region that is allocated as stack right. So, we saw so let us look at the entry dot s file on sheet 10.

(Refer Slide Time: 08:39)

```
Aug 28 14:35 2012: xv6/entry.S Page 2

1050    orl    $(CRO_PG(CRD_0)@P), %eax
1051    movl    %eax, %cr0
1052
1053    # Set up the stack pointer.
1054    movl    $stack + KSTACKSIZE, %esp
1055
1056    # Jump to main(), and switch to executing at
1057    # high addresses. The indirect call is needed because
1058    # the assembler produces a PC-relative instruction
1059    # for a direct jump.
1060    movl    $main, %eax
1061    jmp    *%eax
1062
1063 .comm stack, KSTACKSIZE
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073 INTEL
1074
1075
```

And we will see there is this instruction which is loading this value stack plus kstack size into esp kstack size is a constant. Stack is the symbol that has been declared here right. So, kstack is the symbol in the kernel and recall I am saying that all symbols in the kernel have been compiled to have addresses in the kernel virtual address space which is above kern base right. So, the stack the value of stack will be some you know kern based plus something above kern base and so that plus kstack size, kstack size is let say you know 4096 bytes, so that variable is loaded into esp.

Recall that at this point the paging is already been enabled, so this address should be a valid address. I should be able to dereference esp, because I am operating in this address space right. In this address space esp will be pointing somewhere here. And so, when I dereference it, I will get right value right. So, I was able, so notice that the I have I have only started reading the kernel symbols after I enabled paging.

(Refer Slide Time: 10:05)

```
.text
.globl multiboot_header
multiboot_header:
#define magic 0x1badb002
#define flags 0
.long magic
.long flags
.long (-magic-flags) - 1072
# By convention, the _start symbol specifies the ELF entry point.
# Since we haven't set up virtual memory yet, our entry point is
# the physical address of 'entry'.
.globl _start
_start = V2P_W0(entry)
# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
# Turn on page size extension for 4Mbyte pages
movl %cr4, %eax
orl $CR4_PSE, %eax
movl %eax, %cr4
# Set page directory
movl $V2P_W0(entrypgdir), %eax
movl %eax, %cr3
# Turn on paging.
movl %cr0, %eax
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
```

Before I enabled paging, this code did not read any kernel symbols except that one kernel symbol that was read was entry page dir, but it was converted to its physical address before it was loaded into CR3. In any case CR3 is going to take a physical address. So, it is ok, and physical address is already mapped right. So, this piece of code in entry dot s is very carefully written. There is some amount of tricks involved in this kind of code.

And, this kind of tricks you can only see if you are actually looking at some kind of you know real code right. So, if you know if you find this interesting, you should probably go and single step and see exactly what is happening you know as soon as you turn on paging what addresses become valid, what remain invalid, what were valid earlier and have become invalid now, what were invalid earlier have become valid now etcetera, etcetera right. So, it is interesting ok.

And then you basically, so at this point though you no need to any conversions from v to p right because stack having an address above 2 GB is a valid address, you already in the new address space. Similarly, main is a kernel symbol which will also have an address which is above kern base. And what you are going to do is you are going to move jump to main right. And the moment you do that, you have reloaded your eip with the kernel virtual address which is above kern base. So, you loaded esp with the kernel virtual

address here appropriately, and here you are going to load the eip with the kernel virtual address ok.

So, at this point, so far, my eip and esp were pointing somewhere here, I have reloaded esp to point here and it is standing on solid ground. And then I jump to main, and once again eip now points here. So, now both my esp and eip point here. And from now on I will basically be executing in this space completely because I have forgotten about all my addresses here; anything I dereference from the kernel will have addresses in this space.

So, from now on I just execute in the space right. So, from now on it is just normal plain c code that can execute you know just plain dereferencing should work right. But this kind of tricky code has to be written in assembly, because you know c does not understand this different address spaces and all that kind of stuff, it is very tricky the programmer has carefully done this very carefully done this alright. So, and so we said let us look at, so it is now going to jump to main.

(Refer Slide Time: 12:47)

```
1213 // Bootstrap processor starts running C code here.          1298
1214 // Allocate a real stack and switch to it, first           1299
1215 // doing some setup required for memory allocator to work. 1300
1216 int                                         1301
1217 main(void)                                     1302
1218 {                                              1303
1219     kinit1(end, P2V(4*1024*1024)); // phys page allocator 1304
1220     kmalloc(); // kernel page table                         1305
1221     spininit(); // collect info about this machine        1306
1222     lapicinit(); //                                         1307
1223     seginit(); // set up segments                          1308
1224     sprintf("ncpuid: starting %x\n", cpu->id);          1309
1225     picinit(); // interrupt controller                   1310
1226     ioapicinit(); // another interrupt controller       1311
1227     consoleinit(); // I/O devices & their interrupts    1312
1228     uartinit(); // serial port                           1313
1229     pinit(); // process table                          1314
1230     twinit(); // trap vectors                          1315
1231     binit(); // buffer cache                         1316
1232     fileinit(); // file table                           1317
1233     sinit(); // inode cache                          1318
1234     ideinit(); // disk                                1319
1235     if(!lisp)                                         1320
1236     timerinit(); // uniprocessor timer                1321
1237     startothers(); // start other processors          1322
1238     init2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers() 1323
1239     userinit(); // first user process                 1324
1240     // Finish setting up this processor in mpmain.      1325
1241     mpmain();                                         1326
1242 }
```

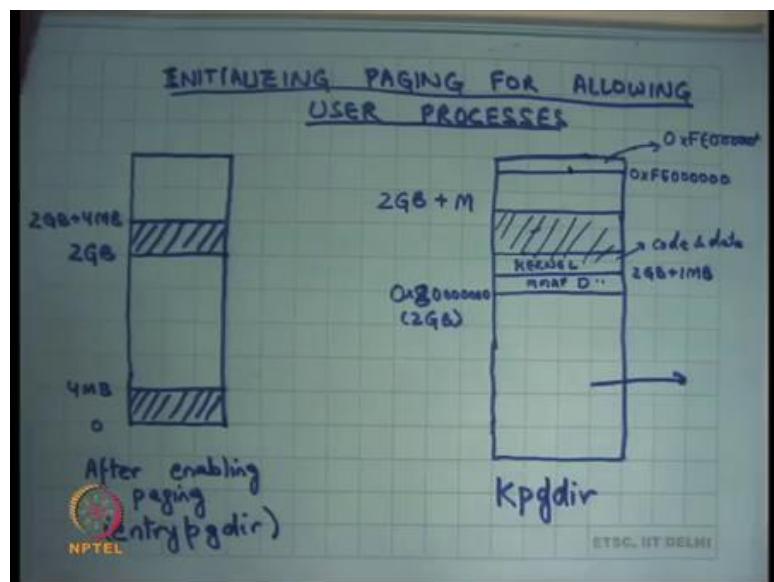
And, here is a code for main right. And main is this long function that you know that we do not necessarily need to go into full detail of. But the important thing to note here is that this is the first instruction that will get executed. And at this point the stack has been initialized correctly right with some size, stack size, kstack size. As long as these functions do not overflow the stack right, do not use too much stack space.

For example, they do not the function called depth is not very large in the above, they do not allocate local variable that are too large or too many local variables that that stack should be sufficient to implement all these to execute all these functions alright. So, the first function it calls this kinit1 one. And so, this is a physical page allocator alright. So, it is going to it going to initialize functions called k alloc and k free that allow you to take physical pages from the available address space. We are going to study this later. But let us just assume that this function works correctly.

And after this function has executed completely, the kernel can make calls to k alloc and k free right. K alloc is just like malloc for kernel. So, just you can allocate memory, and you can free memory except that k alloc only works at page (Refer Time: 14:10). So, malloc can take any size, but k alloc can only k alloc one k alloc call will allocate one page and give it to you ok. And then there is this function k v malloc which will initialize the kernel page table and that is what I am going to discuss today right.

So, we said that this is the page table after the entry page dir and what the kernel wants to do is that it wants to remove this paging this space and map this space completely into physical map address space right. So, right now only 0 to 4 MB region is mapped, you want to map entire 0 to M in this area right. And you want to remove that, so that they can be used for user pages right. Recall that was what the xv6 paging configuration was ok.

(Refer Slide Time: 15:00)



So, in this page kernel page dir, so let me call at K page dir. I want to switch from this address space to this address space, and so what I am going to do I am going to map right now it is just 2 GB plus 4 MB, I am going to map the entire physical memory from here right. So, I am going to say 2 GB plus M and this is entirely going to go to 0 to M in physical memory right, so that is what I want and how I going to do it we are going to look at the code later.

But let us just see what we want right, we want to map the entire physical memory here, also recall that the first 1 MB was reserved for memory map devices right. The first 1 MB of physical memory address space is actually not memory it is you know console and the other things. So, even that gets mapped, but that is just reserved for the memory map devices. Also recall that I loaded the kernel at 1 MB. So, so this is let us say M map let us say M map d memory map devices. Then there is some area which will be loaded for the kernel.

Recall that we had loaded the kernel starting at 1 MB of the physical address space. So, starting at 1 MB, you are going to have the kernel both code and data right. And all the other space I am going to call it free space right, and that is the space I am going to manage using k alloc and k free that is a space I will say this is space that you can use for your heap right or this is the this is the vacant space.

And this is the space that you can allocate for your data structures like the page directory right. So, where is the page directory can go to get allocated. Recall that the entry page directory was a global variable and so that was allocated in the data section of kernel. But all anything other than that, for example, per process page directories right, so all these things are going to be allocated from this extra space which is wherever the kernel ends and from there on till whatever capital M is the physical size of physical memory.

Also, this is the space allocate memory for data structures like process control box right. And most importantly this is the space from where you are going to allocate memory for the user processes themselves alright. So, what you are going to do is you are going to say k alloc going to get some page and you are going to create a mapping in this area for that. So, recall that is how it works basically allocate some pages here. Let us say a user says I want more page, or a user says I want to load a particular executable and that executable is larger than the current allocation of the process.

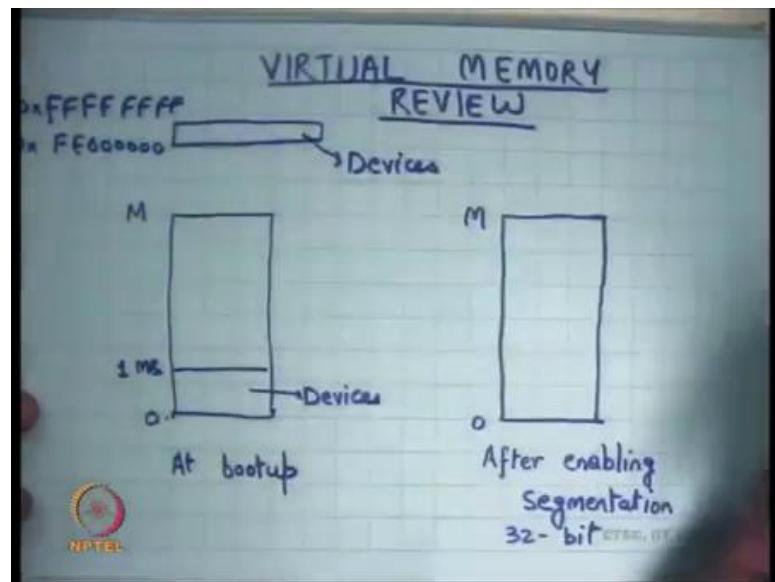
What is it going to do is, it is going to create mapping here it is going to allocate some space of here? Whatever space it allocates from here, it will have some backup in physical memory right some pointer into the physical memory. Some whatever pointer I get from allocation here I am going to convert it into its corresponding physical address, and then create a mapping from here to that physical address right, recall that the entire memory is been mapped here.

So, the entire memory has been mapped in my virtual address space, I allocate some area from that virtual address space, I get a page, I convert the address that I got to its physical address, and then I create a mapping in the user side of things to point to that right. So, these pages which are the mapped in the user side actually have two mappings in the page table: one on the kernel side and one on the user side right. So, if the two names for the same physical location, one name will be kern base plus something, and other name will be whatever the user want it to allocate it wherever it wanted to allocate it right, so.

So, I want to switch from this organization to this organization. Also point out that the top there is a slice of virtual address space on the top that starts at 0XFE and six zeros till 0XFF right. This slice of address space is actually used for memory map devices also. So, just like this area is used for memory map devices, this area is also used for memory map devices, and so this maps identically to the physical address space FE is same thing alright.

So, basically the physical address space in on a 32-bit X86 machine, the physical address 0XFE and six zeros is not actually pointing to memory, it is pointing to some memory map devices. And the kernel wants to retain that access. So, what it does is it just says the top virtual address space maps identically to its corresponding physical address space. So, if the kernel ever wants to access those devices, it can just access it using that address the same address right. So, basically, I am not going to use this top address space from FE000000 to FFFFFF for anything other than memory map devices right they map identically to physical address space right.

(Refer Slide Time: 20:59)



So, just to make this discussion more complete, at bootup the address space was not just this right. Actually at bootup the address space was 0 to 1 MB is devices, 1 MB to M is whatever your physical memory is, and also there is some chunk on the top FE e 2 3 4 5 6 to FF This area is can also be accessed. However, this will also point to devices right in the physical address space, so that was the original physical address space.

So, the amount of physical memory that you can have in your system is not really 2^{32} minus 1, it is 2^{32} minus 1 minus 1 MB minus whatever this is ok. And, so what the kernel wants to do is now retain this access and so it is going to map this identically to the corresponding physical address space right that is all. Anyways that is not really important, but when we are looking at code, it will help us in understanding what is going on ok.

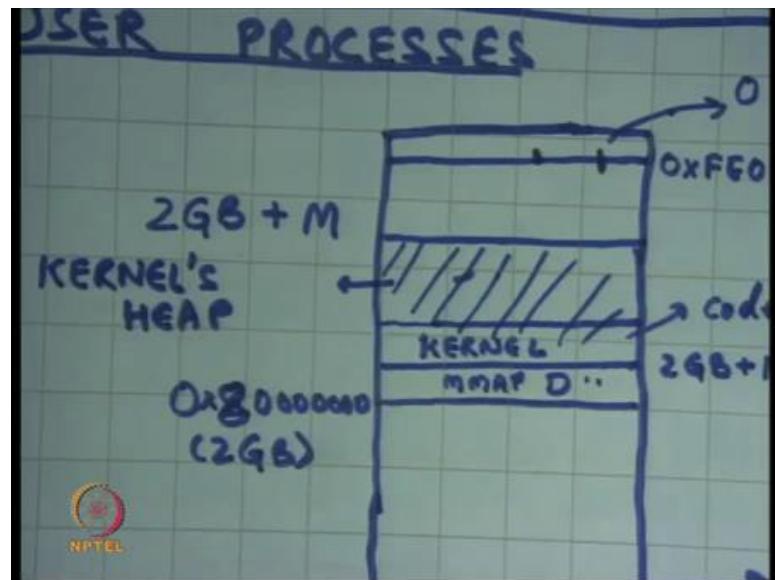
(Refer Slide Time: 22:27)

```
1753
1754 // Allocate one page table for the machine for the kernel address
1755 // space for scheduler processes.
1756 void
1757 kvmalloc(void)
1758 {
1759     kpgdir = setupkvm();
1760     switchkvm();
1761 }
1762
1763 // Switch h/w page table register to the kernel-only page table,
1764 // for when no process is running.
1765 void
1766 switchkvm(void)
1767 {
1768     lcr3(v2p(kpgdir)); // switch to the kernel page table
1769 }
1770
1771 // Switch TSS and h/w page table to correspond to process p.
1772 void
1773 switchuvm(struct proc *p)
1774 {
1775     NPTEhcli();
1776     cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(*cpu->ts));
1777 }
```

So, let us look at what is kvmalloc doing on sheet 17. So, basically at the highest-level kvmalloc is going to initialize a new page table which is going to have this kind of mapping right, and it is going to switch to it. And it is going to forget about the old page table and that is it right. So, the kvmalloc function is just two lines. It calls the function called setupkvm; this is going to initialize a new page dir k page dir which will have this address spaces mapping.

And it is going to return a pointer k page dir to that particular page directory ok. So, it is going to initialize the page directory, where is it going to get the pages for to initialize for this page directory from its heap right from all this area that we discussed. This extra area: this is a kernels heap.

(Refer Slide Time: 23:24)



So, it is going to allocate a page directory from here, and it is going to initialize it, and it is going to return a pointer to the page directory ok. That that return pointer will be a virtual address in the kernel space right because everything in the kernel from now on is in the virtual address. When you allocate a page the return value that you get is also a virtual address. So, if you want to convert it to a physical address you need to subtract kernbase from it 2 GB from it right.

So, this function is going to allocate a page table and return a pointer to it this point is going to be a virtual address. And then I am going to call switch kvm. So, k page dir happens to be a global variable and. So, you just set up k page dir to this and switch kvm is just going to load cr3 with kpgdir except that it is going to call v 2 p on k page dir before it loads return cr3 perfect.

You allocated k page dir in your virtual address space initialized it, you got a virtual pointer you converted into physical pointer and loaded into cr3, recall that cr3 takes only physical pointers ok, so that is very simple. What we are going to look at next is setup kvm how exactly this page table is getting initialized alright. So, what is the structure of the page table.

(Refer Slide Time: 24:59)

```
1727     int perm;
1728 } kmap[] = {
1729 { (void*)KERNBASE, 0,           EXTHEM,    PTE_W}, // I/O space
1730 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text
1731 { (void*)data,   V2P(data),    PHYSTOP,   PTE_W}, // kern data
1732 { (void*)DEVSPACE, DEVSPACE,   0,           PTE_W}, // more dev
1733 };
1734
1735 // Set up kernel part of a page table.
1736 pde_t*
1737 setupkvm(void)
1738 {
1739     pde_t *pgdir;
1740     struct kmap *k;
1741
1742     if((pgdir = (pde_t*)kalloc()) == 0)
1743         return 0;
1744     memset(pgdir, 0, PGSIZE);
1745     if (p2v(PHYSTOP) > (void*)DEVSPACE)
1746         panic("PHYSTOP too high");
1747     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1748         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1749                     (uint)k->phys_start, k->perm) < 0)
NPTEL
```

So, on the same page at line 1737, here is the code for setup kvm alright. And what it is doing is it first calls kalloc function to get a page from the kernel heap right. So, let us just assume that kalloc just allocates a page from the kernels heap and returns the pointer to it right, just like malloc, and puts it into page dir. Just in case the malloc failed or kalloc failed, then it just says oh I cannot proceed, it will return 0 and ultimately it will return 0 and tell the user that you know something has failed. Why could kalloc fail, give me some reasons?

Student: (Refer Time: 25:48).

You have run out of memory. So, let us say the physical memory was very small and so the head room that you have above the kernel space is very small. So, when you call the first kalloc, the first page itself got failed right.

Student: Sir.

Yes, question?

Student: Sir, how is the kalloc for (Refer Time: 26:03), right now we do not have a page directory or anything. So, from where will it allocate the page?

So, there was one function called k init that I skipped alright. So, I am going to discuss the later, but let us just assume that this address space has already been so this right now

we are working in this address space right 2 GB to 2 GB plus 4 MB. So, that is a great question. Assuming that there is some headroom above the kernel in this first 4 MB, I should be able to allocate a page right. So, it is not really limited by the size of physical memory at this point, it is actually limited by 4 MB this artificial limit that we used right, because you have only mapped the first 4 MB.

So, assuming that if the kernel size was so large if the kernel was you know let us say 3.9 MB, then my kalloc would have failed right irrespective of how much physical memory I have. So, if my kernel was indeed that large, I should have mapped more area here in my entry page dir, but my kernel is small right not that large let us say less than 1 MB. So, I have enough space.

So, when you call kalloc, you can actually allocate space from here ok, because yeah you right at this point I am I am not switched I do not have a heap, my heap is actually very small. So, what you have done is he is just initialized whatever space is here to a heap and that is from that is where he is going to serve his request for kalloc alright ok.

So, so he is going to get a pointer in that kernbase to kernbase plus 4 MB space and that is going to be stored in page dir. The next thing it does is it zeros out the page dir. So, the pointer that he gets has a one-page size memory area that can be used right. So, page size is 4096 bytes, and recall that a page directory structure itself was one-page rights 4096 bytes. And so, what it does is just zeros out, so memset page dir zero-page size means zero out the entire page.

Student: (Refer Time: 28:08).

What does it mean to zero out the entire page directory, basically means no mapping exist right, because recall that for a mapping to exist the present bit in an entry should be set right? So, if you zero out the entire thing, none of the present bits are set, and so there are no mappings in this page directory initially ok. We can ignore this, this is just an error check, but then it is going to say for k is equal to k map k is less than this map pages right.

So, it is going to so k map is an array of mappings. It is going to look at this array, and it will call the map pages function to create mappings in the page table for that array.

What this array is k map is basically telling you what the regions are, that need to be mapped. For example, so it basically so static struct k map, it says this is the virtual address at which you need to map a region. This is the corresponding start physical address at which this region should be mapped. This is the end physical address. So, it also tells you the size of the region that needs to be mapped, and these are the permissions with which you should map it right.

So, for example, what we want to do is we want to say 0x you know this kernbase 2 GB value. So, 2 GB value to this 2 GB plus 1 MB should be mapped to 2 GB plus 1 MB with all privileges read-write execute. Then 2 GB plus 1 MB to whatever the kernel code is and read only data is that should be mapped to 1 MB to 1 MB plus whatever kernel codes kernel sizes in read mode. You do not want that the code should be writeable, just a just a precaution measure.

So, you just basically map it in read only mode. And then for everything else which is kernels data, and all the other memory that should be mapped in read-write mode in this area to the corresponding physical address right. And finally, this address 0XFE00 should be mapped identically to the same address in physical memory. So, those are the four sort of regions that you need to map right, four contiguous regions that you need to specify and that is what this array is telling you right.

(Refer Slide Time: 30:31)

```
// This table defines the kernel's mappings, which are present in
// every process's page table.
static struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
} kmap[] = {
    { (void*)KERNBASE, 0,           EXTMEM,    PTE_W}, // I/O space
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},   // kern text+rodata
    { (void*)data,     V2P(data),    PHYSTOP,   PTE_W}, // kern data+mem
    { (void*)DEVSPACE, DEVSPACE,    0,           PTE_W}, // more devices
};

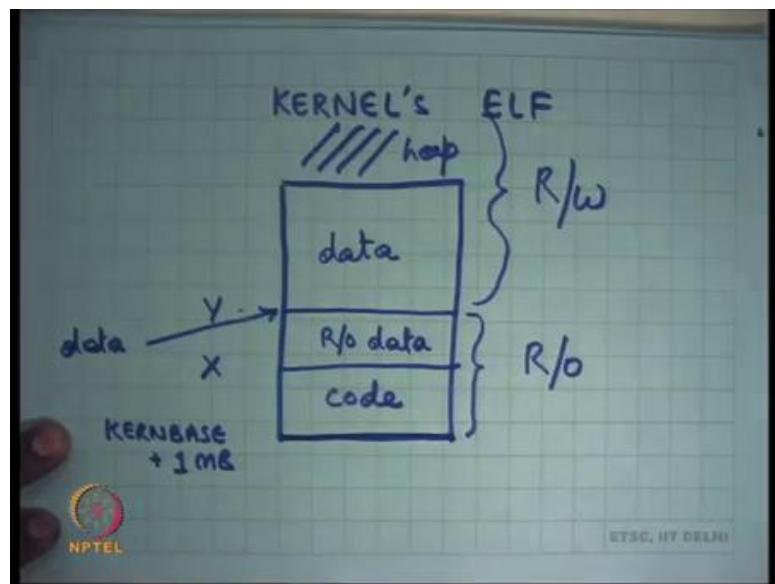
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;

    if(pgdir = (pde_t*)kalloc()) == 0
        return 0;
```

So, the first mapping is saying start mapping at kernbase which is 2 GB. Physical address 0 to physical address 1 MB EXTMEM is 1 MB with write permissions alright. It is just initializing the IO space saying kernbase to kernbase plus 1 MB map it to physical address 0 to 1 MB with write permissions, this is my IO this is my IO space memory map devices.

Then the next thing I have is starting at kernlink. So, kernlink is what, it is going to be the place at where you will start the kernel, it is 2 GB plus 1 MB ok, and V 2 P kernlink is 1 M b. So, recall that we have loaded the kern kernel starting at address 1 MB, the boot set at loaded the kernel starting at address 1 MB. So, we are going to load the kernel starting at 1 MB at address 2 GB plus 1 MB ok. The size the endpoint of this particular segment will be determined by where the data starts.

(Refer Slide Time: 31:49)



So, the kernel has been organized the kernels ELF has been organized in such a way that this is kern base plus 1 MB that is the start address. It will first have some code whatever the size of the code is and so somewhere the code will finish let us call that position X. Then there will be some area which will be called read only data ok. You can specify read only data. For example, in C, if you say const something then declared the global variable that gets allocated in the read only data right.

So, let us say you know data goes at Y, and then everything else is let us say data right. So, there are pointers in the kernel when you compile the kernel, there are symbols in the

kernel which tell you that here is the data, here is read only data and so on. So, this Y is pointing to so that is where the data point is pointing. So, what, what the loader is going to do is it is going to consider this as one segment and map it with read only permissions right. And it is going to map this and everything above it. So, above it is basically heap. After you load it above this is heap. So, everything above is going to be mapped with read-write privileges right ok. So, that is what that is happening here.

(Refer Slide Time: 33:25)

```

1712 // For the kernel's instructions and I/O data
1713 // data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1714 //   rw data + free physical memory
1715 //   0xFe000000..0) mapped direct (devices such as input)
1716
1717 // The kernel allocates physical memory for its heap and for user memory
1718 // between V2P(end) and the end of physical memory (PHYSTOP)
1719 // (directly addressable from end..V2P(PHYSTOP)),
1720
1721 // This table defines the kernel's mappings, which are present in
1722 // every process's page table.
1723 static struct kmap {
1724     void *virt;
1725     uint phys_start;
1726     uint phys_end;
1727     int perm;
1728 } kmap[] = {
1729     { (void*)KERNBASE, 0,           EXTEND,    PTE_R0}, // I/O space
1730     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
1731     { (void*)data,   V2P(data),    PHYSTOP,   PTE_W0}, // kern data+memory
1732     { (void*)DEVSPACE, DEVSPACE,   0,           PTE_W0}, // more devices
1733 };
1734
1735 // Set up kernel part of a page table.
1736 pde_t*
1737 setupkvm(void)
1738 {
1739     pde_t *pgdir;
1740     struct kmap *k;
1741
1742     if((pgdir = (pde_t*)kalloc()) == 0)
1743     {
1744         return 0;
1745     }
1746     memset(pgdir, 0, PGSIZE);
1747     if(V2P(PHYSTOP) > (void*)DEVSPACE)
1748     {
1749         panic("PHYSTOP too high");
1750     }
1751     /* Map all the pages in [0..PHYSTOP] */
1752     for(k = kmap; k < kmap + (PHYSTOP - KERNBASE)/PGSIZE; k++)
1753     {
1754         /* Map the pages in [k->phys_start..k->phys_end] */
1755         /* Map the pages in [k->virt..k->virt + k->perm] */
1756     }
1757
1758     /* Switch to the new kernel page table */
1759     /* For when we switch to the new page table */
1760     /* void */
1761     /* switchkvm(virt) */
1762     /* Switch to the new page table */
1763     /* For when we switch to the new page table */
1764     /* void */
1765     /* switchkvm(str) */
1766     /* pushc3(); */
1767     /* cpu->got[SE]
1768     /* cpu->got[SE]
1769     /* cpu->tsi.ind
1770     /* cpu->tsi.ind
1771     /* void */
1772     /* void */
1773     /* switchkvm(str) */
1774     /* */
1775     /* pushc3(); */
1776     /* cpu->got[SE]
1777     /* cpu->got[SE]
1778     /* cpu->tsi.ind
1779     /* cpu->tsi.ind
1780     /* trc(SE, T55)
1781     /* If(p->pptr)
1782     /* panic("ow")
1783     /* trc(V2P(p-
1784     /* popc3(); */
1785 }
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797

```

Kernlink V2P kernlink which is 1 MB to V2P data right; so, wherever data starts till that point you map it with zero permissions 0 mean read only permissions in this case alright. If it is writeable, then you say PTE W, if 0 that means, read only. And then starting at data map it from at V2P data all the way till phystop, phystop is the size of the physical memory let us say alright or the maximum size of physical memory that you will support.

All that all the way up to phystop, you map it with writeable permissions, so that is kernel data plus all the other memory that will be used as a kernels heap right. And then you map devspace to devspace which is just FE00 identically right. So, that is what you are going to do, you are going to read this array which has these mappings contiguous mappings in a nice readable way.

(Refer Slide Time: 34:27)



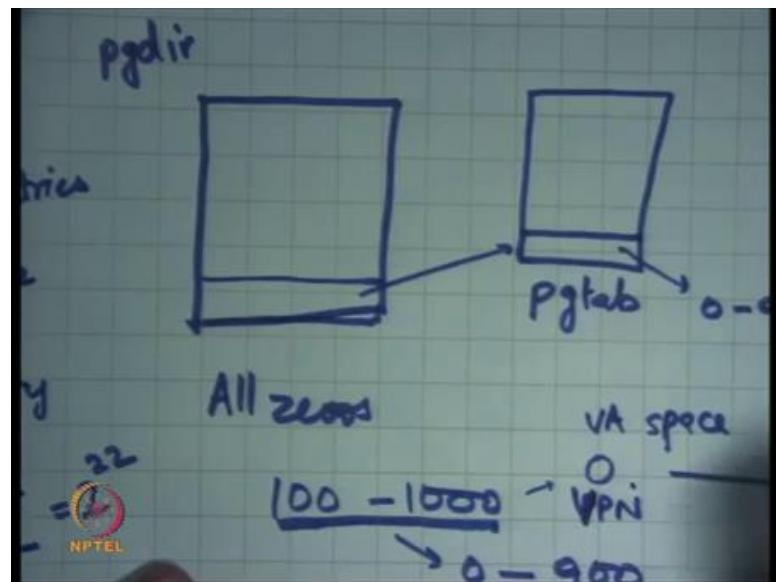
```
1723 static struct kmap {
1724     void *virt;
1725     uint phys_start;
1726     uint phys_end;
1727     int perm;
1728 } kmap[] = {
1729     { (void*)KERNBASE, 0, EXTHDR, PTE_W}, // I/O space
1730     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+odata
1731     { (void*)data, V2P(data), PHYSSTOP, PTE_W}, // kern data+memory
1732     { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
1733 };
1734
1735 // Set up kernel part of a page table.
1736 pde_t*
1737 setupkva(void)
1738 {
1739     pde_t *pgdir;
1740     struct kmap *k;
1741
1742     if((pgdir = (pde_t*)kalloc()) == 0)
1743         return 0;
1744     memset(pgdir, 0, PGSIZE);
1745     if (pgdir(PHYSSTOP) > (void*)DEVSPACE)
1746         panic("PHYSSTOP too high");
1747     for(k = kmap; k < kmap[NELEM(kmap)]; k++)
1748         if(mappage(pgdir, k->virt, k->phys_end - k->phys_start,
1749                     (uint)k->phys_start, k->perm) < 0)
1750             panic("mappage failed");
1751
1752     switchin();
1753     pushell();
1754     cpuregs.gpr[0] = 0;
1755     cpuregs.gpr[1] = 0;
1756     Tir(SEG_T);
1757     if(p->ppg)
1758         panic("ppg");
1759     popcl();
1760 }
1761
1762 void
1763 switchin()
1764 {
1765     pushell();
1766     cpuregs.gpr[0] = 0;
1767     cpuregs.gpr[1] = 0;
1768     Tir(SEG_T);
1769     if(p->ppg)
1770         panic("ppg");
1771     popcl();
1772 }
1773
1774 }
```

Sheet 17

And for each of these mappings, you are going to call map pages right. So, what map pages is going to do is just going to create these mappings. Map pages takes an argument a virtual address, the corresponding size which is given by phys end minus phys start in the k map structure, the corresponding physicals address at which it should be mapped and the permission with which it should be mapped alright.

So, it first takes the page directory in which it should create the mapping, it assumes that the page directory should already created, the virtual address, size, physical address permissions. And, if it succeeds and it will return a nonzero a positive value greater than equal to 0 value, otherwise return a negative value. What are some reasons for why it can fail? Recall that to map pages it may need to create allocate more pages right.

(Refer Slide Time: 35:31)



Right now, you are basically having a page dir. So, this is the page dir that is initially completely 0 doubt, all zeros. Now, let us say I want to say I want to map address 100 to 1000 to physical address 0 to 900. This type of the particular example right; so, I want to map virtual addresses 100 to thousand to physical addresses 0 to 900.

What do I need to do I will say address 100 is represented by entry number 0 right, each entry here represents a 4 MB region right because these are 2 to the power 10 entries mapping a space of 4 gigabytes right. So, each entry, 1 entry is mapping 2 to the power 32 divided by 2 to the power 10 which is 2 to the power 22 address space which is 4, 4 MB right.

And, recall that this 4 MB could be mapped using large pages or it could be mapped using another level of indirection using a page table right. So, in this case 100 is going to be this, the first entry within the first 4 MB. You are going to allocate another page table here; it is called as page table. And, in this you are going to say each entry going to allocate 4 kilo bytes.

So, here you going to create an entry saying map to 0 to 900 right. Clearly, I, I cannot just create this mapping, I need to say what is the corresponding page pages and so I can only do this at page (Refer Time: 37:22). So, instead of 0 to 100 to 1000, I will probably have to say map page 0 in VA space to map page 0 in PA space. So, this is just going to map to 0 in this case by the corresponding (Refer Time: 37:47).

So, to be able to do this, to be able to, so for this kind of mapping, I need to potentially allocate a new page to store this mapping right, and so this allocation could potentially fail. So, page directory is the top-level structure. Recall that a page table is a two-level structure. And so, whatever I want to map I may need to create allocate pages for the second level. And so, if that allocation fails, if I am running out of memory, I can probably fail so that is one reason for example, why map pages could fail ok.

(Refer Slide Time: 38:27)

```

    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern
    { (void*)data,      V2P(data),      PHYSTOP,   PTE_W}, // kern
    { (void*)DEVSPACE, DEVSPACE,      0,          PTE_W}, // more
};

// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;

    if((pgdir = (pde_t*)kalloc()) == 0)
        return 0;
    memset(pgdir, 0, PGSIZE);
    if (p2v(PHYSTOP) > (void*)DEVSPACE)
        panic("PHYSTOP too high");
    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
        if (!map_pages(pgdir, k->virt, k->phys_end - k->phys_start,
                      (uint)k->phys_start, k->perm) < 0)
            NPTEL
}

```

Yes. So, if I want to map let us say 0 to phystop in one go, I will divide it in two chunks and create entries in the page table, because recall that I cannot just the paging system does not allow me to map all at once. I have to you know divide it into either large pages or small pages, and then create appropriate entries in the page table to create that mapping ok.

Student: Already corrupt those pages.

So, the question is that let us say I want to map an entire region from data to phystop in the page table right. Question is how much space do I need to map this? The amount of space that I need to map this is basically whatever these spaces divided by

Student: 4 KB.

4 KB. If you are using small pages right, so that is not that much. So, let us say this space was, so the overhead is basically you know less than 1 percent, or less than 0.1 percent actually right. So, whatever this space was let say this space was x bytes, then the amount of space that you required to make that those x bytes is x divided by 4000 roughly right, so that is less than you know 0.1 percent of x .

Student: But when I map those in the page directory means that I have already also have a backup in my physical memory for those mapping.

Could you repeat?

Student: Like if I map it from my data to physical stop.

Ok.

Student: My in my page directory and it means that in my physical memory also I have allocated space for that particular memory only.

You have mapped that memory; you have mapped that memory. You are not, so there is a difference between mapping and allocation. You have just mapped that memory which means this name is going to refer to this location. You have not allocated that memory right. Allocation means you are going to you have basically committed to using it. You have just mapped it you are going to use it later on, ok. The memory that you have allocated is the kernels code, and kernels read only code and kernels data a kernel's read only data and kernels data.

All that has been already there are some contents into it which are useful which should not be overwritten. So, all that memories mapped, but that is a small piece. Everything above that you have mapped not necessarily used, you are going to use it later. And you are going to use it using kalloc and kfree functions, you are going to manage that using kalloc and k free functions.

Student: Sir, so we do not need to store that mapping means that mapping will require some space.

Which mapping?

Student: What should do physical?

We do not need to store that. So, can you say your question fully?

Student: I am saying that we need some space to store that mapping that this space number in virtual will point to this page number in physical

Sure.

Student: So, how do we do that?

So, how do we say that this page number in virtual space maps to this page number in physical space. We say that using a page table entry right. So, one entry in the page table basically says that this page maps to this page right. And so, an entry of 4 bytes gives you information about mapping for a region of 4 KB. So, it is you know point one percent overhead or space in that sense alright.

So, what map pages is going to do is, it is going to fill in the structure called page dir such that this area in virtual address space gets mapped to this area in physical address space ok. And it is going to do it using small pages alright. So, recall that I had said that you know one common optimization used in mainstream kernels is that you use large pages to map the entire kernel right that save space, but xv 6 does not do that I mean it is not so xv6 just to make thing simple uses small pages to even map the kernel address space. So, all these regions are going to be mapped using small pages right.

(Refer Slide Time: 42:53)

```
// (directly addressable from end..P2V(PHYSSTOP)).  
// This table defines the kernel's mappings, which are present  
// every process's page table.  
static struct kmap {  
    void *virt;  
    uint phys_start;  
    uint phys_end;  
    int perm;  
} kmap[] = {  
    { (void*)KERNBASE, 0,          EXTMEM,    PTE_W}, // I/O s  
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern  
    { (void*)data,     V2P(data),    PHYSSTOP,  PTE_W}, // kern  
    { (void*)DEVSPACE, DEVSPACE,    0,           PTE_W}, // more e  
};  
  
// Set up kernel part of a page table.  
pde_t*  
setup(void)  
{  
    pde_t *pgdir;
```

Annotations:

- Address 0xE000000 is written above the first entry.
- The value 222 is written next to the second entry (KERNLINK).
- The value 224MB is written next to the third entry (data).

In this case for example, if I wanted to be smart about it, I could have probably said look this area is 1 MB, so I am going to use small pages to map this area because you know 4 MB pages are too big to map. This area, this area from kernlink to V2P data, this is also very small I said it is less than 1 MB. So, even this is it does not make sense to use large pages. I am going to use small pages for this, but this area from V2P data to phystop and assuming my phystop is very large right like you know 100 MB or 200 MB.

Then this area is for potentially mapable using large pages right, but we are going to forget about it, we can say we are going to map all these areas using small pages right. And finally, this area is also small, so large pages may or may not help yeah, I mean perhaps large pages will help here too, but in any case, we are going to use small pages to map all this.

Student: Sir, phystop maps the end of (Refer Time: 43:46).

Phystop is just a constant defined in the kernel which basically says what is the maximum amount size of physical memory, that xv6 supports alright. So, phystop is just setup to hexadecimal E 3, 4, 5, 6. I think E and six 0s which is 224 MB right. So, it just irrespective of the size of the physical memory it maps data to phystop identically ok.

So, if the physical memory was larger than phystop those that area is not accessible right, above phystop area in the physical memory will not be accessible. If the physical memory was less than phystop let us say the physical memory was only 100 MB, then you have just unnecessarily mapped extra area, but that is I mean if the user ever accesses that area is going to get some error right, some exception ok.

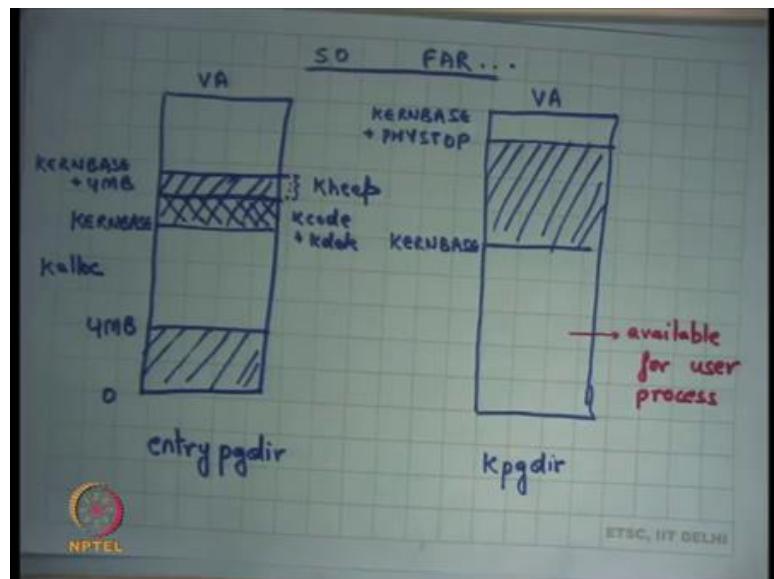
Let us stop here.

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

Lecture – 16
Processes in action

Welcome to Operating Systems lecture 16.

(Refer Slide Time: 00:29)



So far, we have been looking at the virtual address space using paging right. And, we saw that when the operating system boots up, it boots up in the physical address space then it enables segmentation. Then it enables the 32-bit mode without paging in which case still it has the physical address space it can access the physical address space and at some point, it enables paging right.

And, the first page when it enables paging for the first time, it enables it using this page directory called entry page dir right and at the entry page dir you have an address space which is look something like this from 0 to 4 MB, you are mapped to physical address 0 to 4 MB. And, from KERNBASE to KERNBASE plus 4 MB which you know which is starting at the which is where the kernel is mapped. Let us call them these call these addresses the high addresses.

So, at the high addresses starting at the high first high address to the 4 MB high address, they are again map to the same. So, the both these regions are aliased to the same physical memory area right in the entry page directory. And, we saw that this is a nice way of doing things because this allows you a very smooth transition from a flat; from a address space which only had this much right and then so it the initialization code still runs in this space assuming that the kernel itself its within 4 MB. And, then at some point the kernel moves it stack and itself the instruction pointer in this space right.

And from then on it will want to always execute in the high addresses leaving the low addresses available for the user process right. So, eventually you will want to move from the entry page directory, which was just a temporary thing so, something like this where you have mapped the physical memory in the kernel address space. So, starting from KERNBASE to let us say the maximum physical address that that the kernel; suppose is phystop. So, you will map KERNBASE to KERNBASE plus PHYSTOP here; map to the physical address and the lower addresses will be available for user process right. So, the user can map its code here, map its data here and so on, yeah yes question.

Student: What happens to the entry page directory when the kernel page directory is created?

When the kernel page directory is created what happens to the entry page directory ok? One answer is that only in the entry page dir this remains, and you know; so question is does the entry page dir get deallocated right or does it keeps consuming the space that it was consuming earlier right. After all entry page dir is living in physical memory, it is consuming some space right.

Initially I had some use for it; so, I moved to entry page dir and I used it for something and then I have moved to k page dir and so what happens to the entry page dir? Do I de allocate that space and reuse that for something else.

Student: Sir, we could have used the entry page directory for the kernel page dir from; like we could have overwriting the phys entry page directory which is turning with the kernel page.

Here is the suggestion how about just overwriting entry page directory with the contents of kernel page directory and that; that would have saved some space alright. So, based on

the codes we have seen so far for xv6; what do you think? How was entry page directory declared? It was a global variable right. It was a global variable which means it is taking some static space and it can never be deallocated right. Also it has a fixed size and its contents are sort of also you know initialized in some way, you can overwrite them and one way to overwrite them could have been that you just overwrite them with pointers right.

On the other hand, kp page directory was not a global variable in fact, it was allocated from the kernels heap. In fact, it was allocated from this region from KERNBASE to KERNBASE plus 4 MB. So, anything that you allocate from heap; the nice thing is you can de allocate it and you can start using it for something else. In the code that we have seen for xv6; entry page dir was not allocated of the heap right because by the time the entry page there was needed, the heap was not even initialized.

So, you use the entry page dir has some static array, global array. So, that much space is actually going to get wasted after entry page dir is not used right in the case of xv6 at least right. You may you if you really care about this amount of space which is you know 4 kilobytes, then you would maybe want to you know allocate entry page dir also of the heap and then deallocate it so you can use it for anything else you like later on or you could use it reuse it 4 kpgdir; all these are options.

You know one has to at some point make a tradeoff between coding simplicity and efficiency and so it is to just say that you know let us basically allocate this to a global variable; later I am not going to use it, but does not matter alright ok. So, this is the k page dir and yeah there is another question.

Student: But could not just we have changed the PTE underscored p flag to 0 and that was it could have been deallocated automatically.

The question is could not I have just changed the PTE underscored p flag to 0 and that would have deallocated it automatically. Well, I think there is some confusion between mapping and allocation right. So, mapping basically means that I have mapped, and I have created a mapping between a virtual address and a physical address that is what mapping means. Allocation means I have some bookkeeping to do, so I have said that this amount of space is being used already and this amount of space is not being used; so that is just bookkeeping right.

So, mapping and allocation are two different things so, but when I say I am going to free in a memory area; I am basically saying I want to deallocate it; un mapping it is not going to deallocate it right. When I want to deallocate it basically means that if I want to allocate something else; I can use reuse this space that is what deallocation means. So, that that space should be reusable that is what the allocation deallocation means right; mapping un mapping is something different alright; so, this is kpgdir alright.

Now, I will point out a few things one interesting thing here recall that most of our programs assume that on the null address or 0 address is an invalid address right. If you ever dereference in a null address that basically means it is an error right; that is a convention right.

So, you cannot dereference the 0 address; another convention is when you say malloc and if it returns 0, basically means that the allocation failed. It basically means that the address 0 should really be invalid; if address 0 was valid then you know this kind of a mechanism will not work. So, typically the convention is you know initially at this point notice that 0 is a valid address actually. So, at this point if malloc; you know if this if this area was part of the heap and malloc return 0, it may actually not be an error it may actually be giving you an address which is 0 right, but at this point 0 is not mapped.

So, at this point you know; so as a convention typically, operating systems do not map the zeroth page; just as a convention right that makes your coding interface simpler because you just saying that one particular address called 0 is not going to be a valid address; that is a convention that is being used across the software stack right. If the operating system ever violates that convention, then you know a malloc could actually turn a 0 as a valid page and that would; that would violate my conventions right.

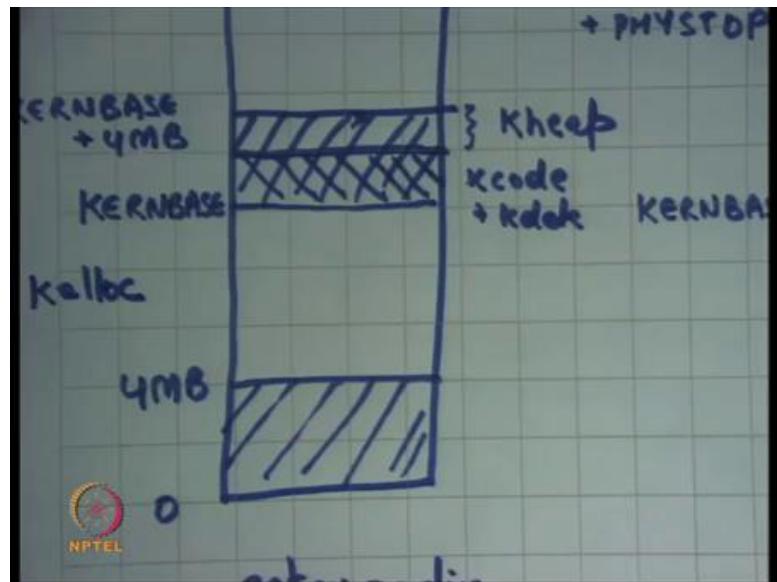
So, as a convention you basically say that at the 0, I am not going to map anything. So, even when the operating system is going to map the user pages; one simple convention that it follows as it is not going to map the zeroth page alright. So, and we were looking at xv6 and we said let us look at the code which basically does all this alright.

(Refer Slide Time: 08:58)

```
1211 extern char end[]; // first address after kernel loaded from ELF file
1212
1213 // Bootstrap processor starts running C code here.
1214 // Allocate a real stack and switch to it, first
1215 // doing some setup required for memory allocator to work.
1216 int
1217 main(void)
1218 [
1219     kinit(end, P2V(4*1024*1024)); // phys page allocator
1220     kvmalloc(); // kernel page table
1221     mpinit();
1222     Tapicinit();
1223     seginit(); // set up segments
1224     cpprintf("\nCPU%d: starting xv6\n\n", cpu->id);
1225     picinit(); // interrupt controller
1226     ioapicinit(); // another interrupt controller
1227     consoleinit(); // I/O devices & their interrupts
1228     uartinit(); // serial port
1229     pinit(); // process table
1230     tvinit(); // trap vectors
1231     binit(); // buffer cache
1232     fileinit(); // file table
1233     finit(); // inode cache
1234     ideinit(); // disk
1235     if(!ismp)
1236         timerinit(); // uniprocessor timer
1237     startothers(); // start other processors
1238     kinit2(P2V(4*1024*1024), P2V(PHYSSTOP)); // must come after startothers()
1239     userinit(); // first user process
```

So, let us look at the code which transitions from entry page dir to k page dir and we said that this function called kvm. So, firstly this function called kinit 1 initializes the physical page allocator alright.

(Refer Slide Time: 09:20)



So, what does it do? It basically says now let us say you know at this point I am executing in entry page dir and what I am going to do is I am going to finish and let us say some area of this 4 MB is used for; this is let us say used for the kernel code plus data k code plus k data right. And all this area above it is going to be used as kernels

heap; let us call it k heap right. And this is where you are going to allocate; you are going to create you are going to use this space to allocate space for k page dir for example, alright.

So, this is the area from where you are going to allocate pages for k page dir. So, before you can do that you need to initialize the space in some way and so you are going to initialize some data structure which is going to be which is which is going to be called the free list and you are going to add all these pages in this area to the free list ok. And so, once you do that; you can use these functions called you know k alloc.

So, once again this is just bookkeeping; so you have you have a you have an internal data structure called free list and k alloc is going to take a page from that free list and return it to you and k free is going to push it back to the field free list alright. So, this statement here this function called kinit 1 is just initializing the heap starting at the kernels end. So, end is basically wherever the you know it is initialized the; so, symbol end itself is initialized by the linker that you know in the in your compilation stage.

So, the end was a symbol that was given to you by the linker which basically marks the end of the code and data of the kernel. So, that is the; that is where the heap starts where the code end data end and till 4 MB right P 2 V of 4 MB. So, KERNBASE plus 4 MB that is what this means. So, end to 4 MB is where you are going to initialize the kernels heap and so what you going to do is kinit 1 is going to push all this to some free list. And, now you whenever you say k alloc is going to allocate a page from that area and you say k free is going to push back a page into that area available for us right.

And so, kvmalloc is going to initialize k page dir and inside kvmalloc; it is going to call k alloc to basically allocate these pages for the page table right alright.

(Refer Slide Time: 12:02)

```
1756 void
1757 kvmalloc(void)
1758 {
1759     kpgdir = setupkvm();
1760     switchkvm();
1761 }
1762
1763 // Switch h/w page table register to the kernel-only page table,
1764 // for when no process is running.
1765 void
1766 switchkvm(void)
1767 {
1768     lcr3(v2p(kpgdir)); // switch to the kernel page table
1769 }
1770
1771 // Switch TSS and h/w page table to correspond to process p.
1772 void
1773 switchuvm(struct proc *p)
1774 {
1775     pushcl1();
1776     cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1777     cpu->gdt[SEG_TSS].s = 0;
1778     cpu->ts.ss0 = SEG_GDATA << 3;
1779     cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1780     lcr3(SEG_TSS << 3);
1781     if(p->pgdir == 0)
1782         NPTEL_Panic("switchuvm: no pgdir");
1783     lcr3(v2p(p->pgdir)); // switch to new address space
```

So, now let us look at kvmalloc in that sheet 16; we were looking at it last time but let us look at it again sheet 17 ok. So, kv malloc does nothing it for it allocates a page table in the virtual address space and assigns the pointed of it or the page directory to k page dir.

And, then it calls switch kvm which switch kvm does nothing, but loads the physical address corresponding to the k page dir which is just whatever k page dir is minus kern base and puts it into cr 3 right that is what that is what kvmalloc is going to do; it is going to allocate a page directory and the switch to it.

(Refer Slide Time: 12:41)

```
1726     uint phys_end;
1727     int perm;
1728 } kmap[] = {
1729 { (void*)KERNBASE, 0,           EXTMEM,    PTE_W}, // I/O space
1730 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
1731 { (void*)data,   V2P(data),    PHYSTOP,   PTE_W}, // kern data+memory
1732 { (void*)DEVSPACE, DEVSPACE,   0,           PTE_W}, // more devices
1733 };
1734
1735 // Set up kernel part of a page table.
1736 pde_t* pde_t*
1737 setupkvm(void)
1738 {
1739     pde_t *pgdir;
1740     struct kmap *k;
1741
1742     if((pgdir = (pde_t*)kalloc()) == 0)
1743         return 0;
1744     memset(pgdir, 0, PGSIZE);
1745     if (p2v(PHYSTOP) > (void*)DEVSPACE)
1746         panic("PHYSTOP too high");
1747     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1748         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1749                     (uint)k->phys_start, k->perm) < 0)
NPTEL
Sheet 17
```

Let us look at what happens when you allocate a page directly. So, let us look at setup kvm. So, this is the code for set of kvm; it basically allocates a page of the heap right recall that k alloc which going to give me a page from between end and 4 MB right end and V 2 p 4 MB is going to give me a page from there.

The address of the page will be a virtual address I am only dealing in virtual addresses at this point. If for some reason the k alloc failed you just say you know there is a failure; I am running out of space and this can happen if your kernels code plus data was actually very large it was close to 4 MB let us say right.

Then you memset page dir to 0 which means you say that none of the entries in the kernel page directory at present and then you say you know you do some sanity check that PHYSTOP should be greater than DEVSPACE let us ignore this. And then it iterates over this array called k map which has which contains information about the regions of kernel which need to be mapped and then calls map pages on those regions right.

Last time we looked at the different regions that were mapped; we said you know KERNBASE. So, the first at first feel of this is the virtual address at which you need to map; the second field is the physical address. So, the virtual to physical address mapping is KERNBASE gets mapped to 0; EXTMEM is the size. So, EXTMEM is basically representing 0 to 1 MB space which has all the devices in it and all that.

So, that is mapped with permissions writable then KERNLINK is mapped to V2P KERNLINK to. So, this is kernels text and read only data which is mapped with read only privileges. Then from data to V2P data, data to PHYSTOP, you map it as writable privileges. So, all the write writable data and the kernels heap is in this segment from data to PHYSTOP; we right data to PHYSTOP data to data plus PHYSTOP basically what data do V P 2 V PHYSTOP; PHYSTOP is expressed as a physical thing it is a size; so you basically have it have to right.

And finally, there is this extra thing for DEVSPACE which is basically to keep the upper map memory mapped devices for yeah. So, we can safely ignore this right and so what this loop is going to do is going to look at each of these entries. So, it is going to iterate over the entries of k map and for each entry, it is going to create mappings in the page table right.

So, map page is going to create a mapping in page directory starting and virtual address virt of size phys and minus phys start right; phys and virt is the third field and phys start was the second field starting at physical address phys start with permissions perm right that is what map page is going to do. And we already know what map page is going to do its going to fill in. So, we have initialized a 0; a completely 0, 0 out page dir I am going to fill in these entries some of those entries are going to be non-zero from now on right. Also, I am going to use only small pages to do this.

(Refer Slide Time: 15:46)

```

1672     return &pgtab[PTX(va)];
1673 }
1674
1675 // Create PTEs for virtual addresses starting at va that refer to
1676 // physical addresses starting at pa. va and size might not
1677 // be page-aligned.
1678 static int
1679 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1680 {
1681     char *a, *last;
1682     pte_t *pte;
1683
1684     a = (char*)PGROUNDOWN((uint)va);
1685     last = (char*)PGROUNDOWN(((uint)va) + size - 1);
1686     for(;;){
1687         if((pte = walkpgdir(pgdir, a, 1)) == 0)
1688             return -1;
1689         if(*pte & PTE_P)
1690             panic("remap");
1691         *pte = pa | perm | PTE_P;
1692         if(a == last)
1693             break;
1694         a += PGSIZE;
1695         pa += PGSIZE;
1696     }
1697     return 0;
1698 }
1699

```

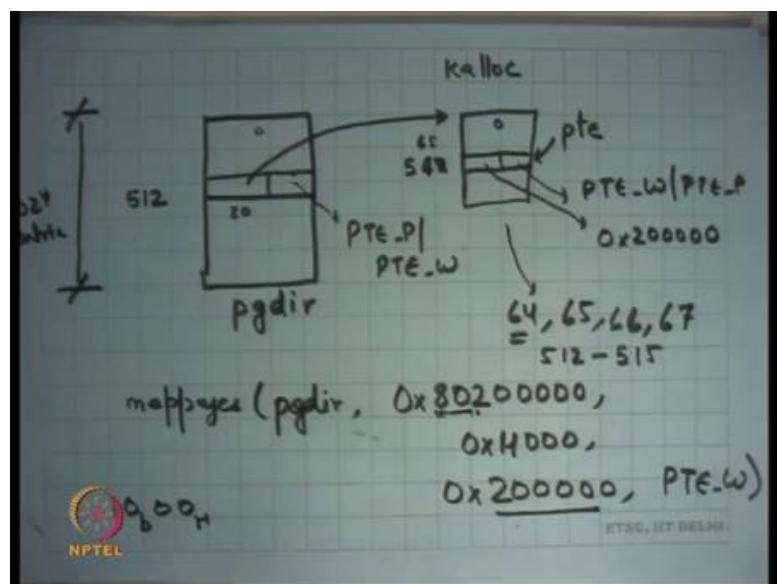
So, this is map pages. So, I am going do use only small pages to map this address space and so not only do I have to fill up the page directory; I have also have to allocate in the second level page tables right and so and point the page directory to the second level page tables and fill those page tables appropriately also right that is what I am going to do.

So, this is simple map pages takes a page directory, a virtual address, size, physical address and permissions and it creates these mappings. Internally, it calls the function; so, it basically says where do I have to start? I start at you know you will round whatever the virtual address is you round it down to the nearest page boundary and that is where you start. And this is where you end you say VA plus size minus 1 and you round it down to its page boundary and that is where you end. And these are the pages that you need mean to map; starting from a to last right. And then you basically say let us walk

the page directory with this address a which means you look at you; you say look at a, you look at the top 10 bits of a; index using that value into the page directory and so on right in the next take 10 bits into the page table and so on right.

And based on that you get a page table entry right and the and then you basically set up the page table entry to point to the physical address that you want to say with their right permissions and the present flag on alright and then and you and then you keep doing this right.

(Refer Slide Time: 17:13)



So, basically if I want to draw this what is happening is, I started with a page directory which was completely 0 right; all everything is zeroed out. And then I said let us say map pages, page dir let us say I wanted to map 0 x 8 0 2 right to physical address and size let us say right; let us say size of hexadecimals 4000 and starting at physical address and let us say permissions are writable right.

If I wanted to do this; what I'm going to do is I am going to say 8020000 this is the virtual address; let us look at the top 10 bits of this address. So, the top 10 bits are going to be you know the first two hexadecimal characters make up 8 bits and the 2 bits of the third one make it you know the top 10 bits and so that is that basically comes up to this end the 512th entry let us say right; it has 1024 entries and all. And this particular address is basically saying that the 512th entry needs to be mapped.

At this entry I am going to see it is a 0 right; so, nothing has been mapped here. So, what I am going to do is I am going to say k alloc and I am going to create a new page. I am going to allocate a new page right from the heap that you have seen before same heap we are going to use k alloc and I am going to point the top 20 bits to the physical address of this page right and I am going to set the permissions here to present and writable right. So, allocated a page a point I converted its into its physical address put it in its 20 bits and also zeroed it out. So, it is also initially 0 completely right.

Then I am going to look at the next 20 bits of this address and so the next 20 bits are going to be let us say.

Student: Next 10.

Next 10 bits sorry I look at the next 10 bits which are what 0 1 0 0. So, 1 0 in binary and let us say 0 0 in hexadecimal which is right; so that that will be probably be it and so that is 60 into 4 16 into 4 let us let us say this is 64th entry maybe I am wrong, but let us just assume that is the 64th entry here.

So, everything else is going to be 0 with a 64th entry; the pointed to the 64th entry is will be called the page table entry pointer. And I am going to set that to point to what? The top 20 bits are going to point to this address right 0x200000 and the flags are going to say PTE W and PTE P right and that is it. So, you created a mapping as desired from 80200, but you also have to do it for all these pages. So, how many pages there these are? These are to the part 12; these are 4 pages right. So, you have to do it for entry number 64, 65, 66 and 67. So, you have to create 4 entries in the PTE right.

So, you allocated one-page table page and you created 4 entries in the page table page to create this mapping. So, in this case the entries 64, 65, 66 and 67; I am going to be created I am assuming that 64 is the correct number I have not done the calculation, but wherever it starts ok. Somebody says its 512 good. So, if this is 512 then it is going to be 512 to 515 that will get created in this ok. So, that is what map pages and is going to do for you.

And we can now look at the code and convince ourselves that it is doing exactly what we wanted. So, walk page dir returns a pointer to the page table entry that needs to be filled up and you just fill up the page table entry with the physical address.

So, in other words walk page dir is going to return a pointer to this entry and you just fill it up with this value. So, walk page dir is going to create return a pointer to this particular entry and you are going to fill it up with the physical address and the flags right. So, that is what it is doing walk page dir returns a pointer to PTE and you just fill it up star PTE is equal to pa and permissions and present flag and you keep doing this right.

(Refer Slide Time: 23:02)

```
1650 // Return the address of the PTE in page table pgdir
1651 // that corresponds to virtual address va. If alloc!=0,
1652 // create any required page table pages.
1653 static pte_t *
1654 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1655 {
1656     pde_t *pde;
1657     pte_t *pgtab;
1658
1659     pde = &pgdir[PDX(va)];
1660     if(*pde & PTE_P){
1661         pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
1662     } else {
1663         if(!alloc) || (pgtab = (pte_t*)kalloc()) == 0)
1664             return 0;
1665         // Make sure all those PTE_P bits are zero.
1666         memset(ptab, 0, PGSIZE);
1667         // The permissions here are overly generous, but they can
1668         // be further restricted by the permissions in the page table
1669         // entries, if necessary.
1670         *pde = v2p(ptab) | PTE_P | PTE_W | PTE_U;
1671     }
1672     return &ptab[PTX(va)];
1673 }
1674
1675 // Create PTEs for virtual addresses starting at va that refer to
1676 // physical addresses starting at pa. va and size might not
1677 // be page-aligned.
1678 static int
```

Walk page dir what is it doing? It is just doing exactly what we did on paper right which is just walk the page table. So, it indexes the page directory using the top 10 bits of the virtual address, looks at that entry if that entry is already present then it gets a page table address and it returns the PTE address by in looking at the next 10; entries bits of the virtual address.

If it is not present already then it allocates it then it uses k alloc to allocate a page for the page table itself and initialize it to 0 and returns the corresponding entry.

So, this is called lazy allocation of the page table; you did not up allocate the page table entirely upfront, you allocated it lazily as in when you were mapping the virtual addresses you started allocating the page table pages. And that is the reason a two-page two level page table helps right that is the whole reason why a two level page table helps because you do not have to allocate the entire page table up front. What is the size of a full-page table?

Student: 4 MB; 4 MB.

4 MB because they are going to 4 kilobyte 4 k page tables 4 k second level page tables and each page table will be 4k in it in size right and so actually it is going to be more than 4 4 MB it's going to be 4 k into 4 k that 16 MB right. So, the sorry there going to be there going to be 1 k page second level page tables each of them 4 k; so that is the only 4 MB of second level page tables plus 4 kb of the page directory.

So, 4 MB plus 4 kb, but you do not need to allocate upfront you do it lazily depending on what virtual address is get map. Assuming that the size of the virtual address is that get that get mapped as small then you are going to basically save some space alright; so that is fine ok.

So, this walk page dir function is basically doing in software what the hardware would have done on every memory access right. The hardware does the same thing actually when you say a virtual address you looks at the top 10 bits, indexes the page directory then index is the page table and so on except that there is one extra complication that you if the page table is not present here you are also allocating it lazily right.

So, walk page dir is just a software counterpart about the hardware would do to walk the page table at runtime ok.

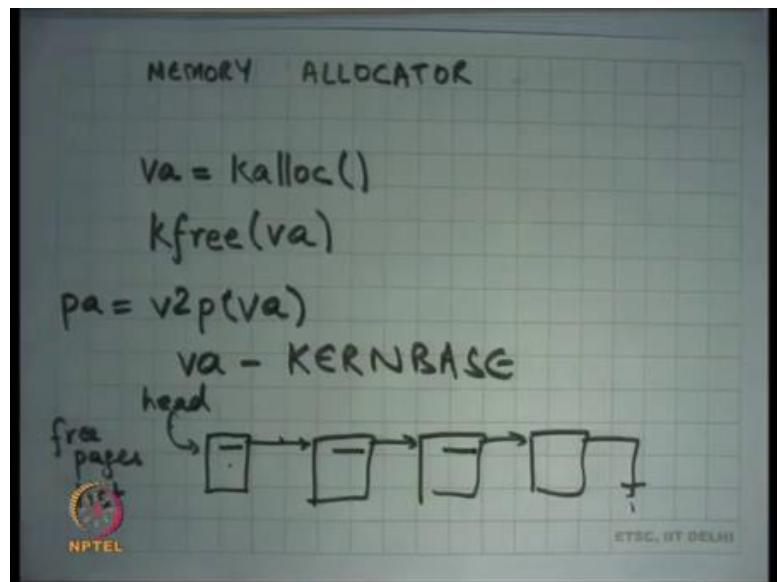
Student: Sir, walk page status walking just the first level not the second one.

It is walking both levels.

Student: Sir when it is not going (Refer Time: 25:27) entry.

See it is actually. So, it is de referencing the page directory to get the page table and then it is returning the appropriate point offset inside the page table as a page table entry right. So, it is it's not; it is not dereferencing the page table entry, but it is getting the address of the page table entry and then sending it back as a return value and so the caller as supposed to dereference the page table entry. So, it is walking both levels alright good.

(Refer Slide Time: 26:06)



So, now let us look at how the memory allocator works right. So, how does the memory allocator work? Recall that we have been using this function called k alloc and what it returns with to me is the virtual address right.

And then I have there is another function that I am using that is called k free which takes a virtual address and freeze it right. In this case, now this is similar to malloc and free that you may have used in your programs; there only difference is that k alloc does not take a size, it just allocates one page right. Also the other thing it has is it does not always return an address that is page aligned; it does not just allocate a page anywhere, it allocates a page at an address that is page aligned right.

So, whatever is the return value of k alloc; the last 12 bits of that address will always be 0. Why does it why do you think a kernel needs an allocator that always returns a page aligned address?

Student: For allocating this.

For yeah, so for a lot of the things that a kernel needs to do; it needs to allocate pages right and then create mappings for them in the page table right. For example, it wants involved pages for the page table themselves the page table recall had to be page aligned right. Similarly, if it wants to create pages for the gives the process it will need to create mappings for it in the page table. So, for a lot of things you need page aligned addresses

and so you know it is easier to basically just have an alligator that always release gives you page aligned addresses right ok.

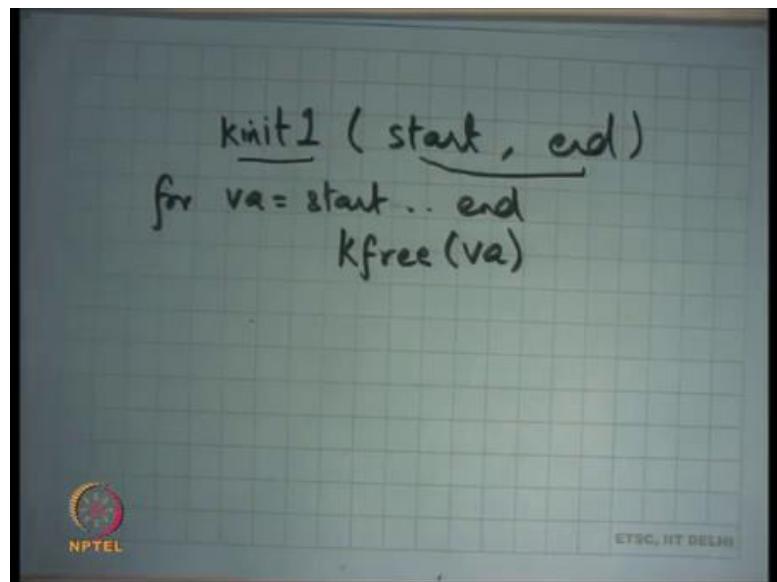
So, these functions always give you virtual addresses and if you ever wanted to convert them to physical address; all you need to do is say v 2 p of v a is going to give you the physical address right and v 2 p is nothing, but v a minus KERNBASE; that is a nice thing about the organization that we have I just sort of follow them from that.

So, how does, how this typically be implemented? Well basically the internally the kern maintains a list of free pages, just a linked list of free pages right. And then it is; it is pointed to by some head pointer and when you say alloc it just takes the first entry from here and updates head to the next point here and just returns that address right and when you say free it just adds it to him. Also notice that the next pointer itself is stored within the pages themselves right; you do not need extra space to store the next pointer because these pages are not getting used anyways. So, you can just use the space within them to store the next pointer itself ok.

So, this is you know this is roughly how malloc and free also work except that malloc and free need to be more complicated because they need to allocate variable sizes of data. So, you need to worry about fragmentation issues, but if you if you are just doing the fixed size allocator, all you need to do is just maintain a list and use the next pointer itself is going to be stored in the list ok. And so what is going to happen is that initially k when you said kinit 1 adds all these pages to the free list basically what it does is it just calls k free on all the add pages present in the space right.

So, just keeps calling k free on all the pages in this space, recall that this is the kernel heap and so all these pages need to be added to the free list. So, one way to of adding all these pages to the free list is just called keep calling k free on all these pages right and so all are them get added to the free list and now you and now your allocator and deallocator work using k alloc and k free.

(Refer Slide Time: 30:33)



So, basically k init; the function k init 1 simply just you know takes a start and an end address and just you know iterates over this address and each page that it sees in that address you just adds it to the free list using k free. So, it just says k free v a for and let us say for v a from start dot end right. Now of course, I am simplifying some things here start and end need to be page lined addresses and all that right. So, if you look at the look at the implementation of k init 1, you are going to roughly find something like this right.

Let us look at k; let us look at the calls to k init 1. So, so k init 1 free adds all the pages starting at end till P 2 V of 4 MB and adds them to the free list right that is what k init 1 is going to do and at this point you have some free list so that you can satisfy some k allocators and. So, kvmalloc and all the functions here are going to use the space which is within the 4 MB. So, notice that all these functions are going to be using only the heap which is lower than 4 MB right, any allocation all the space above 4 MB in the physical address space is not being used at all at this point right.

(Refer Slide Time: 32:06)

```
16 INC
17 main(void)
18 {
19     kinit1(end, P2V(4*1024*1024)); // phys page allocator
20     kmalloc0; // kernel page table
21     mpinit0; // collect info about this machine
22     lapicinit0;
23     seginit0; // set up segments
24     cpprintf("\nncpuId: starting xv6\n\n", cpu->id);
25     picinit0; // interrupt controller
26     ioapicinit0; // another interrupt controller
27     consoleinit0; // I/O devices & their interrupts
28     uartinit0; // serial port
29     pinit0; // process table
30     tvinit0; // trap vectors
31     binit0; // buffer cache
32     fileinit0; // file table
33     iinit0; // inode cache
34     ideinit0; // disk
35     if(!ismp)
36         timerinit0; // uniprocessor timer
37     startothers0; // start other processors
38     kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
39     userinit0; // first user process
40     // Finish setting up this processor in mpmain.
41     /* no:
42 */
43 }
44 
```

NPTEL

Then there is another function called k init 2 that finally, makes available the other physical memory also and what it does is it just says free all the memory starting at 4 MB P 2 V of 4; 4 MB which is you know KERNBASE plus 4 MB to P 2 V of PHYSTOP.

So, add all these pages from starting at 4 MB till PHYSTOP also to the free list and at this point you know your k alloc will also start taking pages from space above 4 MB ok. Why is there a must come after start others? Well there is a diff; so, k init 2 relies on some you know initialization of locking subsystems etcetera. So, k init 2 requires; so because xv6 is a multiprocessor operating system; the same memory allocator serves requests to all the CPUs and so there needs to be some locking inside the allocator right because if two peoples simultaneously access it there should there should be some issues.

So, we are going to look at it in more detail, but for now let us just assume that k init 2 requires to initialize the locking subsystem and to be able to do that it needs that other processor should have started alright and after it has allocated all this memory; so it at this; so why PHYSTOP? Is it ok to say PHYSTOP? What if; what will happen if the physical memory, so PHYSTOP is just a constant right in the in xv6 code and PHYSTOP just is a constant which is around 240 MB right. What if the amount of physical memory on the machine was less than PHYSTOP? Is it to do this?

So, how many say no ok? 5 people. How many say yes? 0 others are undecided ok. So, well it is not right because recall that the free list is actually get there, there is data that is being written in the free list also right the next pointers and so you need to dereference those values to be able to write anything on there that and if there is no physical memory there then dereferencing about generator generate an error an exception right.

So, in a way xv6 is lamely assuming that the amount of physical memory will always be greater than you know PHYSTOP. On the other hand, if the amount of physical memory was greater than PHYSTOP; is that a problem? No, that is no problem because you just using less than the amount of available memory right. Recall that you could I mean has a difference between mapping and allocation.

Even though if the amount of physical memory was less than PHYSTOP, I said it was to map addresses right because I just created a mapping; as long as I do not dereferences those mappings it is ok, but once I start using them that is the problem right. And I can only use those places if I add them to the heap right that is only that is only way, I can sort of start using them and then de referencing them and all that.

So, it was its not to use a memory less than PHYSTOP if you are because of the statement here for xv6 yes.

Student: Because if we consult incorrect address then make a hardware (Refer Time: 35:30) as just calculate modulo the after size and get (Refer Time: 35:36)

So, question is if you generate an indirect in incorrect address incorrect physical address right. So, they are two different things one is the program can generate an incorrect virtual address which means that the address was not mapped in the fulfill virtual page right. So, there is a certain type of exception that is gets thrown at that time, but if the page is actually mapped and the physical address is actually computed, but when you actually go on the bus and ask the memory for that physical address; the memory says oh I do not have that physical address. So, what happens right. so, it is an exception that gets generated ok.

The alternative that was suggested by you is why can kern the address get wrapped or anything of that sort no I mean wrapping is not possible assuming that you are only

dealing with 32 bit addresses and this is 32 bit bus there is no wrapping that can happen really alright.

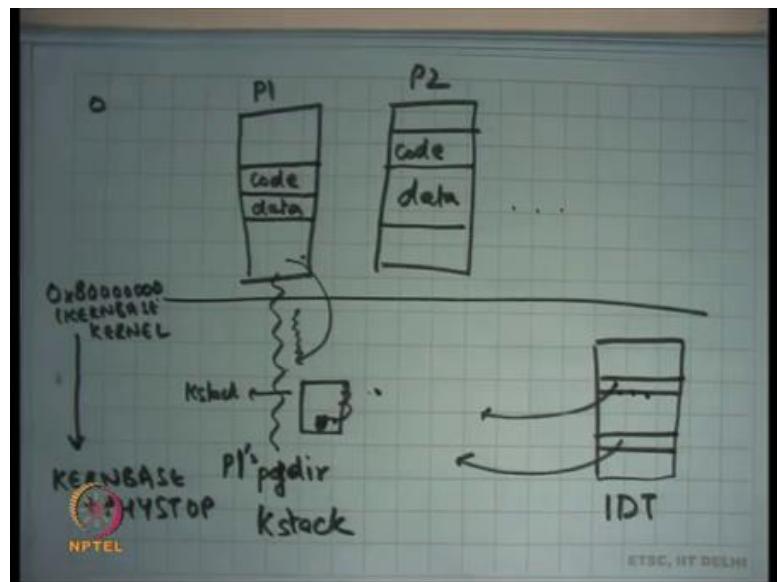
And after that I am going to call user init which is going to initialize my first process and after that my processes are going to run just like we have talked about in the Unix world, where they are the first process is going to get started that is the init process. And, the init process has some you know hard coded code in it which says let us say for the first process. And, let us say the first process is init process folks the shell process and the shell process basically takes commands from the standard input and depending on what the command is it may 4 more processes and so on ok.

So, that is how it typically works that is how it typically works in Linux also there is one init process that starts up and it then reads some files and depending on that it starts up some processes that should be started up at the beginning, some of those processes are going to be let us say the login process the x window system or whatever right.

So, notice that the first process is unique because that is the process that is created by the kernel, all other processes are not created by the kernel they are forked by already existing processes that is the only way to create new processes right. This is the first process that gets created by the kernel and that is what the user and it is going to do ok. So, I am going to look at user in it in the next lecture, but before we do that let us review how this how the system is going to run from there on right.

So, we have initialized everything, we have mapped entire physical address space if started the first process; the first process is going to make system calls and I have also initialized my device of system. So, they are going to be timer interrupts and all that kind of time right and so let us look at you know a sample execution of what is going to happen what is going to keep happening actually right.

(Refer Slide Time: 38:15)



So, so firstly, let us say this is the kernel and this is process P 1 which is the first process P 2 let us say it forked another process and so on right I am I am drawing it inverted. So, let us say this is address 0x8 right or KERNBASE right and so these addresses from here to here which is let us say KERNBASE plus PHYSTOP will be magnified the kernel right and everything from 0 to KERNBASE is the user side of things.

So, a kernel will have let us say its own code somewhere here and we will have some data each process right and similarly he may have its code here and this data here and they will have some heaps and all that hm.

And also there will be something called an Interrupt Descriptor Table we have seen this before IDT that is going to have pointer to which space right; they are going to have pointers to handlers and all pointers will be in the kernel space right. So, let us say P 1 wants to make a system call it uses a software instead of instruction; the IDT gets consulted the system called handler gets called the, but the system.

So, in the context of the same process in the same page directory; so, while this process is executing a certain page directory has been loaded into the hardware; when it makes a system call you switch from user to kernel when you use the same page directory. So, in other words you are basically using the executing in the context of P 1 at this time. We are executing in the kernel, which is shared across all processes video always exists, but at the time of the system call itself you are executing in the context of the process that

made the system call. Because why because you are using the same page directory you have not change the page directory right; we just change the instruction pointer.

Also, so you know you are using P1 page directory and you are using P 1's kernel stack. Assuming it is a process model each process has a different kernel stack. So, you are running one P1's k stack right and how does the hardware load the P 1 k stack? Using the task state segment that must have been set by the OS maybe we know that alright.

So, it is going to execute in P1's k stack it is when it executes in P 1 k stack, the first few things it is going to do is its going to save all the registers of the executing process in the k stack itself right. So, let us say this is the case stack; the first few registers are saved by the hardware namely c s, e i p, s s, e s p, e flags right. The first five registers are saved by the hardware and the next few registers you can save in software which is which is how xv 6 will do it and this is how most operating systems will do it. It will just say push this is just to push that, register though this is a have not be modified. So, you know the values are still preserved and you just keep pushing in them on the k stack.

Once you have pushed all those registers you have saved the state of the user process, we are going to execute some logic on behalf of the process which may involve some privileged operations. And then you are going to say let us return back right and returning is going to be just pop off those values that you had saved on entry one by one, some of those values are going to be popped off in software. And, the last five values are going to be popped off in hardware by using the i rate instruction right and you are going to execute back in the process mode, user mode.

How does the process give arguments for the system call? One way that it just sets up its resistor values to indicate the arguments right it. In fact, sets up the resistor value so also indicate the system call number and how does the kernel give a return value to the system call?

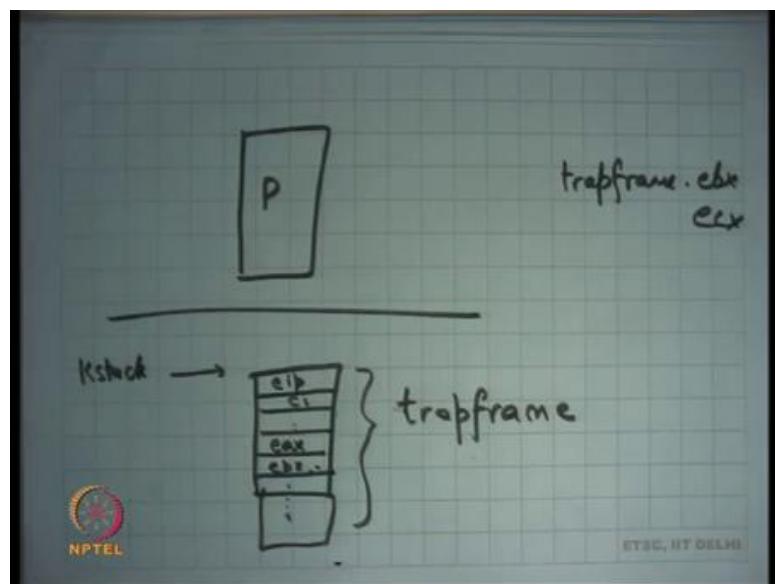
Student: Using the eax register.

Using the eax register so, but what does it need to do here?

Student: Modify.

Modify the saved value in the stack. So, whatever the state value in the stack of the; at the place where eax is living, you modify that. So, at the time of return it is going to get popped off into the eax register and the user is going to see the new eax value right. It does it is not good to just change your own eax because you have own eax is going to get lost when you return to P a. You need to change the eax that is saved in the stack at the right offset and so when you are going to turn back that eax is going to get visible to the user alright.

(Refer Slide Time: 43:41)



So, this structure; so let us say this is a process P when it executes here it starts at k stack right it saves eip, cs and so on and it also saves other registers like eax and software ebx and so on right. This structure which lives on the stack is called the trap frame; while I am executing in the kernel, I can look at the trap frame to look at my arguments and also return a return value right.

So, the first thing a kernel does is create at the entry the first thing the handler will do its going to create this trap frame. The first five entry of the trap frame have already been created by the hardware, the next whatever number of entries depending on the number of registers in the architecture; you going to create them in software by hand and now you going to execute the logic.

So, for example if you wanted to look at your arguments you are going to look at the trap frame and look at the value stored and so let us say you know trap frame dot ebx create a

contains argument number 1 to the system call right ecx contains argument number 2 edx contains. So, it is not the real register that is containing it is the save register which is in the trap frame that contains the arguments of the system call and that is where you also write your value to give a return value to the system call ok.

Student: Sir, what is the upper limit on the arguments directly (Refer Time: 45:18).

What is the upper limit on the number of arguments that can be passed to a system call? That is defined by the OS designer right and clearly the number of arguments cannot be unbounded because you know you have a finite amount of kernel stack. Typically, you know on kernel on Linux you would have you know at most 5, 6 arguments if you wanted more arguments then you will use that use pointer chasing to do that alright. So, it is possible that one of these arguments is actually a pointer that points into some structure which lives in this address space ok.

And so, the kernel can look at this pointer, dereference it and get more arguments from there right. And, your lab 2 is actually going to expose you to this kind of argument passing from the user to the kernel for system causing is going to implement this argument passing ok. So, on a system call to control move from here to here; the trap frame gets set up, the kernel executes on behalf of the process in the same address space at the process and then at some point it returns and the return is just unwinding of the stack at some point you are going to unwind the trap frame.

And, unwinding of the trap frame means you just load the registers where the trap frames values and then you call iret and you are back in the user space. The code the logic can read the trap frame and right to the trap frame for arguments and return values. Same thing happens if there was an external interrupt; let us say there was a timer interrupt that goes up right.

So, if there is a timer interrupt that goes off, that fires then the execution moves from user space to kernel space the some call the trap frame gets saved, some logic gets executed. Let us say the logic that gets executed is the logic or scheduler that at this point decides that you do not want to let this process continue running and you want to switch to another process; in which case you will switch the address space, switch the kernel stack and unwind it from there ok.

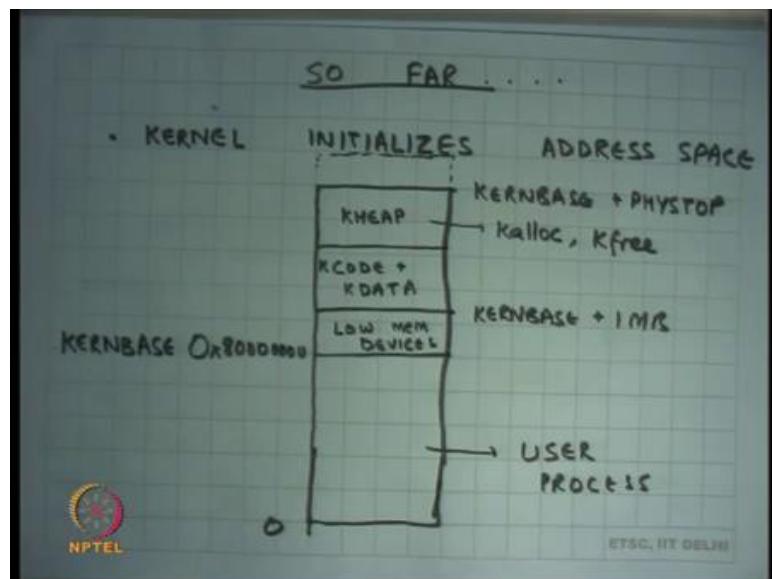
We are going to discuss this more next time.

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

Lecture – 17
Process structure, Context Switching

Welcome to Operating Systems lecture 17.

(Refer Slide Time: 00:31)



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

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

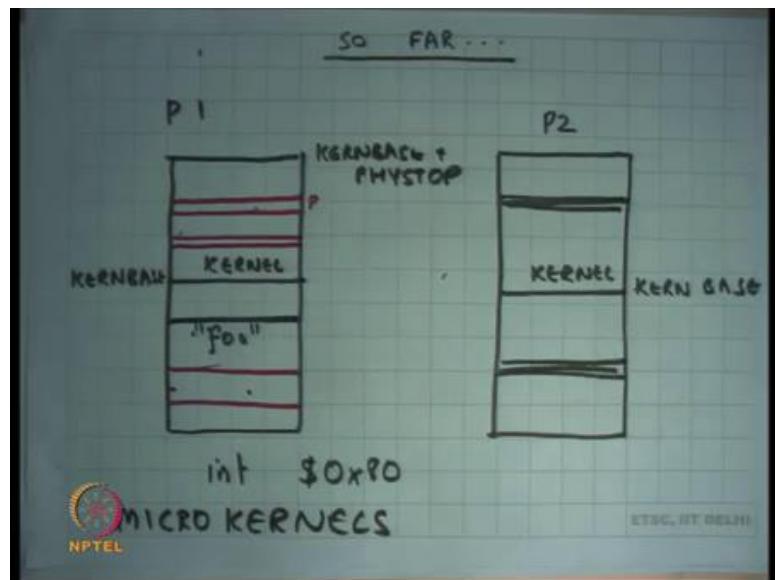
is adding that back page back to the free less so that it can be used and so, memory can be reused and that way you can sort of not run out of memory.

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

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

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

(Refer Slide Time: 02:55)



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

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

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

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

So, by ensuring that only the pages which belong to the user and ever mapped in this area the kernel basically ensure that the user cannot touch any privilege memory or the user cannot the process cannot touch another process memory alright. Also, we said that if once again if a process needs to make a system call then some code here is going to

execute the interrupt instruction let us say it says int some number which indicates the system call number on Linux it is the hexadecimal 80 number.

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

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

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

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

So, this is the huge optimization I mean this is a big advantage of doing things in this way as opposed to doing things in a way you were using segmentation for example, because; segmentation will change the address space for the kernel or using a separate page table for the kernel right. So, you know often you may have wondered why I am eating up so much of space for the kernel in the virtual address space that way I am

basically squeezing out the process right a process could ideally we had as big as a 4 gigabyte space on 2 to the power 32 bit machine.

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

Student: Sir.

Yes question?

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

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

Student: Yes.

Yes why?

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

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

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

mean to answer your question it is possible for the virtual address space to be bigger than the physical address space.

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

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

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

Student: Sir.

Yes.

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

Ok. So, the question is could not the user. So, I said that the user can plus pointers to the kernel, it can set up some area in it is own memory, it can initialize some values in it is own memory and use the address of that memory location and pass it to the kernel and the kernel can dereference it and because the kernel is executing the same address space it is to dereference. And the question is if the kernel was executing in a different address

space could the user have figured out what the pointer is in the kernels address space to the location that it has set up and pass that value to the kernel that is very difficult right, how will the user know, what is the value of the kernels pointer in that address space right.

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

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

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

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

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

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

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

Student: P directory.

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

Student: How does?

Why does not need to know?

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

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

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

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

Student: Sir, but it is it understood.

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

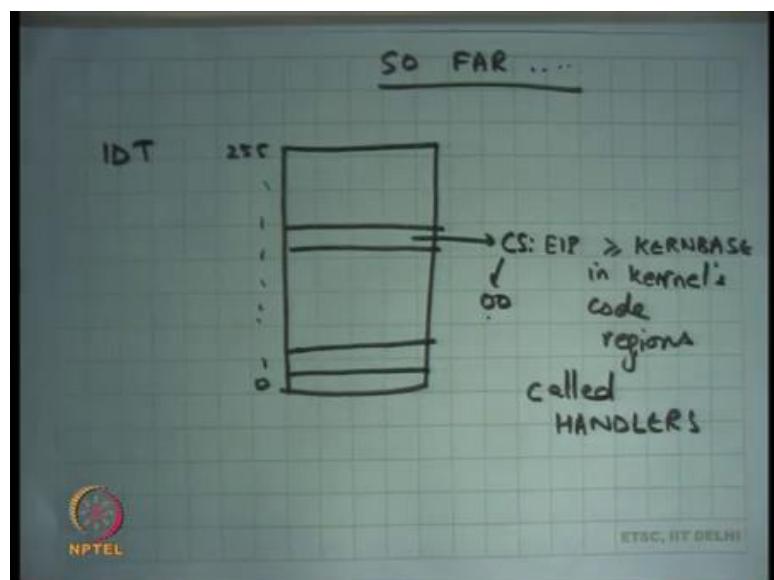
Student: Sir, user page in the allocated from kernel heap. So, why in the system like Linux why do we need 3 GB for user address space and only 1 GB for kernel address space?

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

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

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

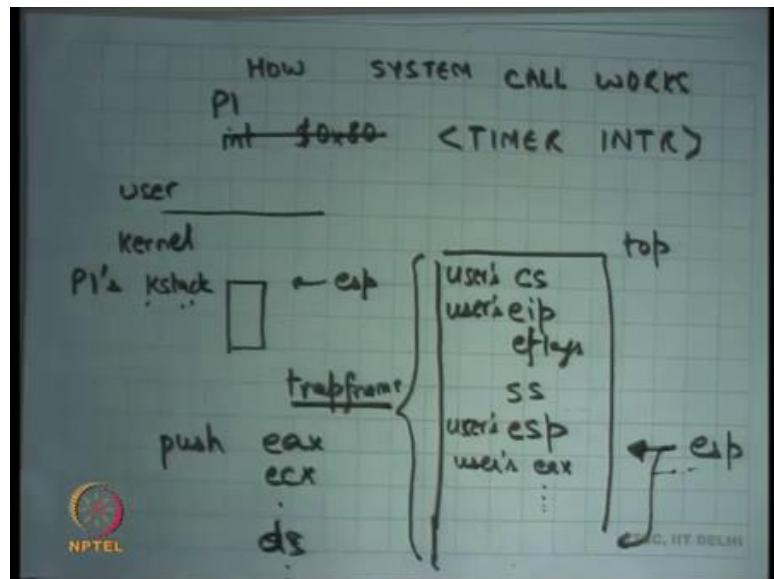
(Refer Slide Time: 17:23)



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

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

(Refer Slide Time: 18:03)



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

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

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

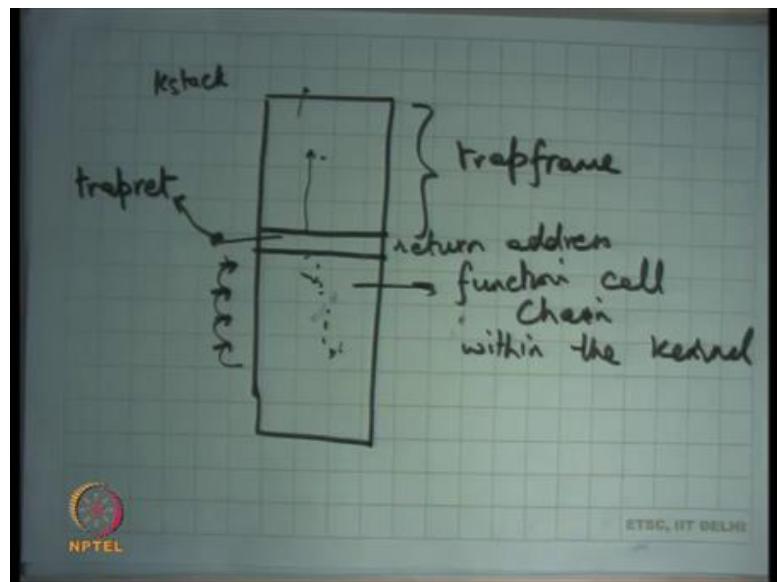
So, now the handler gets called so, at this point the handler gets called and the first few thing it is going to do is it is going to say you know execute instructions like push other register let us say push eax ecx, you know other segment registers let us say ds and so on right. So, it is going to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

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

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

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

(Refer Slide Time: 22:13)



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

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

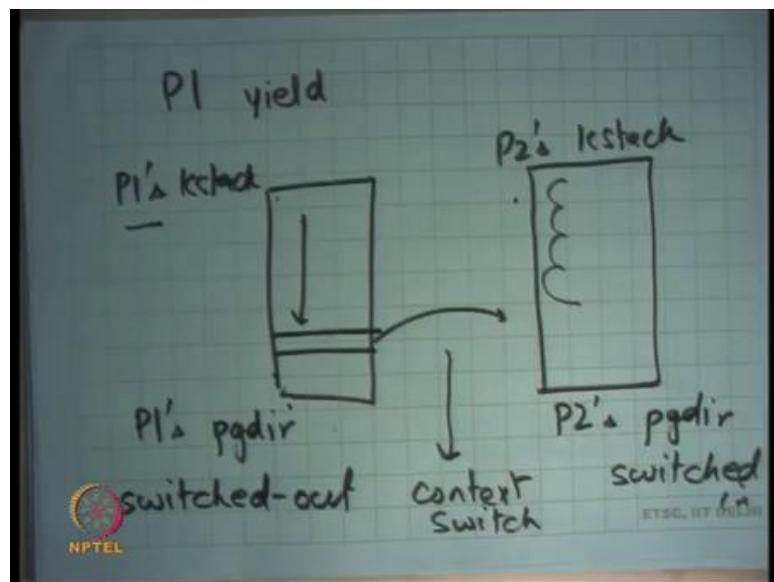
So, on xv6 this function is called the trapret function right. So, in other words what should happen is that whatever is the function call that you called the last return address should be trapret. So, when the last function returns it should return to trapret and trapret will basically just pop off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

right, will stack basically acts as a function as a function stack and so, any every time you return you basically pop off in a value from the stack and jump to it right.

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

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

(Refer Slide Time: 25:49)



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

called a context switch right. So, basically when a process calls yield that is a P 1 calls yield P 1's kstack will get formed the entire chain will get saved on the kstack.

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

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

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

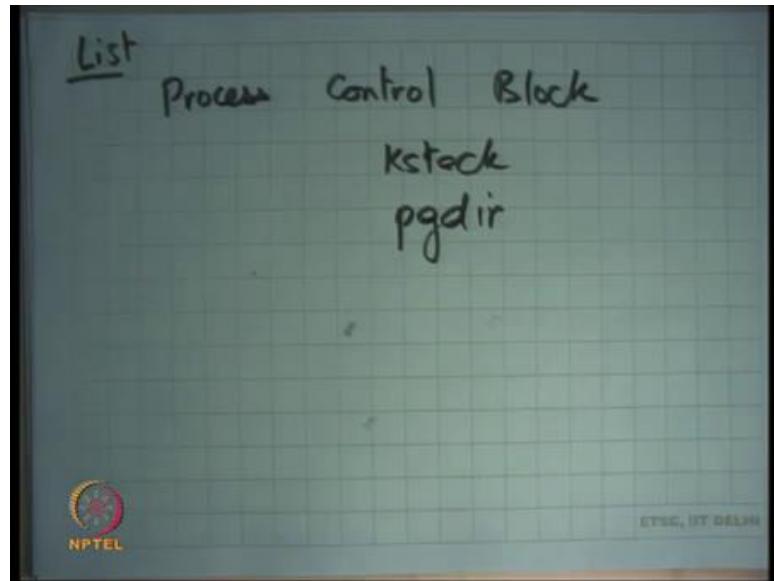
So, that is how a context switch works one process make a system call you execute in the kernel you form some state in the kernel which is saved in the kstack, you switch the kstack, you switch the page directly, you unwind the kstack of the other process. And your back you are suddenly executing exactly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

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

Student: The switch that list is PCB s.

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

(Refer Slide Time: 29:51)



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

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

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

So, you would make sure that the top of the kstack that you initialize that you put inside the TSS the trusted segment is not such that you overwrite that page directory. So, you can use one you know that top plus one area to basically contain the page directory also

right. Is it a big win? Perhaps not, because I mean what are you saved you know you still using the same amount of space overall you could have saved it in the PCB you have saving it in the stack perhaps complicating things a little bit yourself right.

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

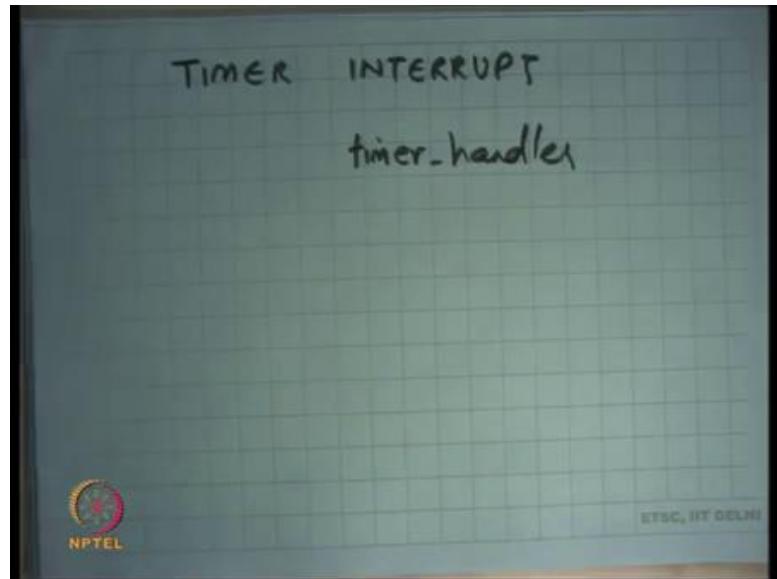
Ok. So, here is an interesting thing, because each process has the same mapping of the kernel everywhere when you context switch the page dir it is right. So, the nice thing about this picture here is that in this context switch when you shifted from P1 page dir to P2 page dir, because you are executing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

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

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

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

(Refer Slide Time: 33:45)



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

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

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

And, then you context switch just like before right you will context switch the another processes kstack would be saved, the other process may have been preempted by a timer interrupt or may have been preempted because of yield call in either case the unwinding

process is similar and you go back the user right, that is how an involuntary context switch happens. The previous case was a voluntary context switch we have somebody called yield this is an involuntary context switch where the time interrupt occurs and the scheduler decides that you need to be switched off or you have been running for long.

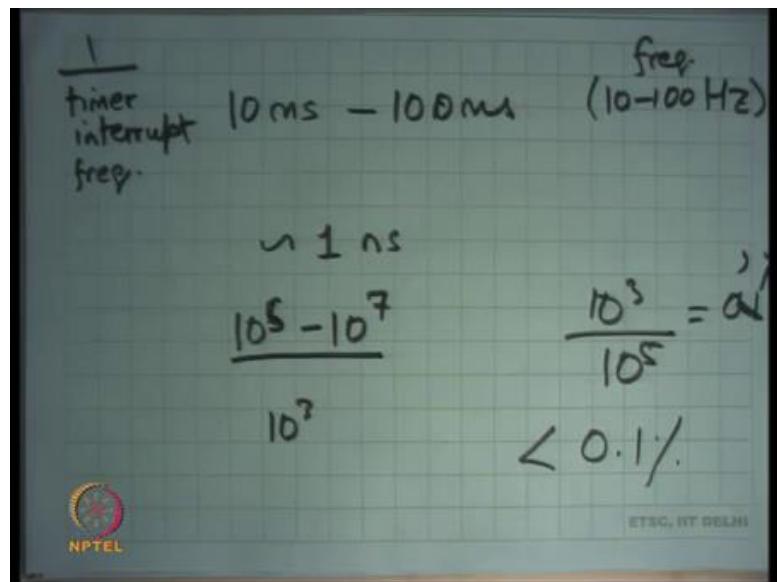
Student: So, the handlers are same for both.

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

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

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

(Refer Slide Time: 37:33)



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

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

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

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

So, somebody says 100.

Student: 20.

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

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

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

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

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

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

Firstly, let us understand that this timer interrupt is being generated per processor right. So, every processor has a separate timer device attached to it in theory right, how it works in physical hardware is the different thing, but let us just assume that each CPU

has a separate timer device attached to it, each CPU is doing its own scheduling basically right and of course, the scheduler is a common scheduler that understands that there are many processors in the system.

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

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

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

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

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

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

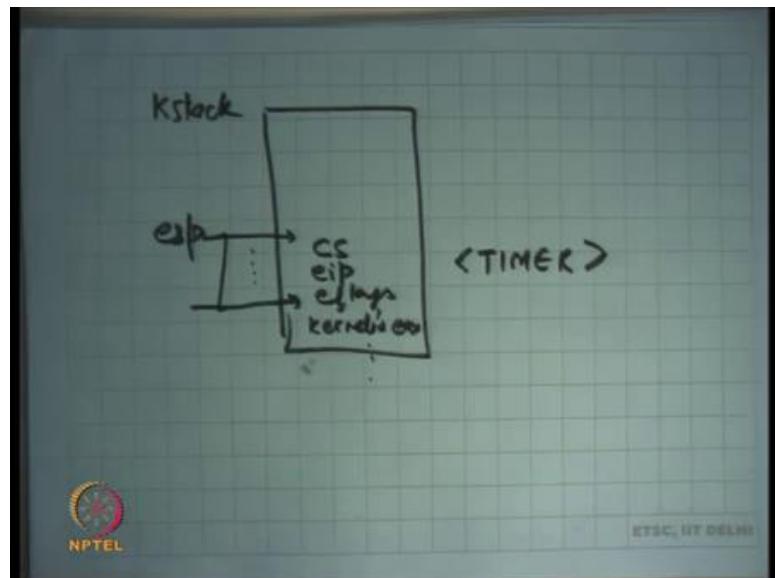
So, I gave you an example that P 1 was running and the timer interrupt occurred in the user mode right and so, this all this started happening after that right, but can a timer

interrupt occur while the process was all processor was already executing in the kernel mode.

Student: Yes

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

(Refer Slide Time: 44:07)



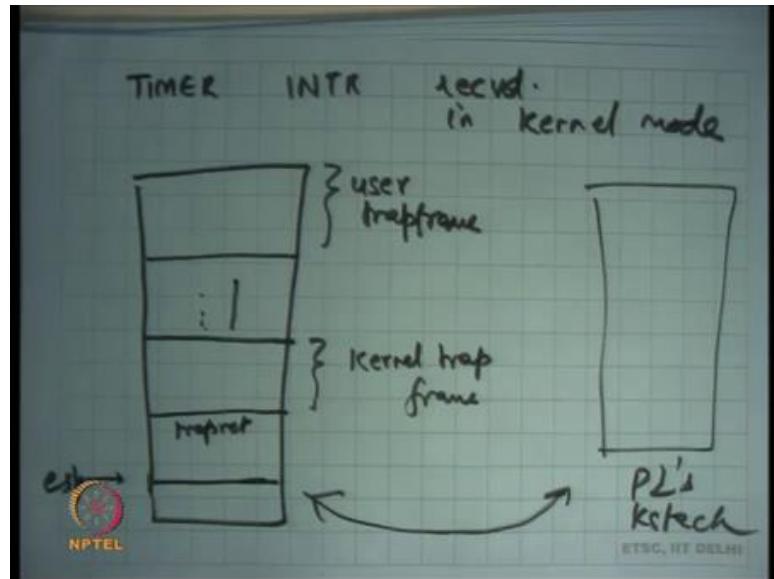
So, if you are executing, let say let say this was a kstack and let say this was a esp while you are executing in kernel mode and a timer interrupt comes let say there is a timer interrupt right. So, what will happen is, esp will just get decremented and cs, eip and eflags of the kernel at that time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

So, kernel's ex kernel is so, this time you are going to save kernels ex and all the other resistors on the stack right. So, you going to save all these things and then you may want

to call the schedule function just like before right. So, the only difference between the previous case was that you started at the top and all the values that you were saving that you were saving were user's values.

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

(Refer Slide Time: 45:41)



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

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

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

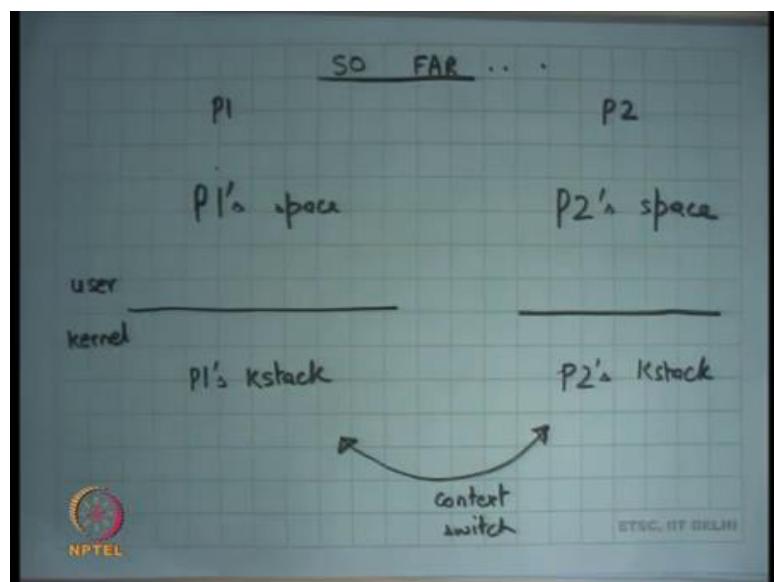
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

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 processes P 1 and P 2. P 1 has, each process has a user space and a kernel space, the user space is private. So, this is P 1 space; this is P 2 space, and the kernel space is shared right. So, this is so the kernel space is common. So, P 1 and P 2 both share the same kernel space.

The only thing that distinguishes two processes are the kstack. So, P 1 kstack is different from P 2'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 these information is basically stored in the kernel stack.

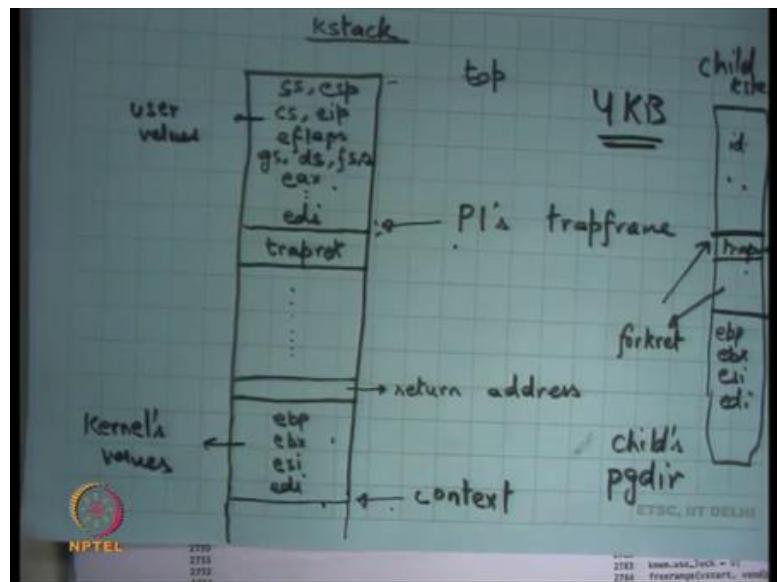
In general, I mean recall that we had made a statement that every user process is also kernel thread right. So, user process the state of the process involve also involves the contents of the address space, because each process has a different address space and different contents. The state of a thread 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 P 1 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 e a x 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 k alloc 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 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 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 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 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 ignored. What will happen if the interrupt gets ignored, 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 up 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.

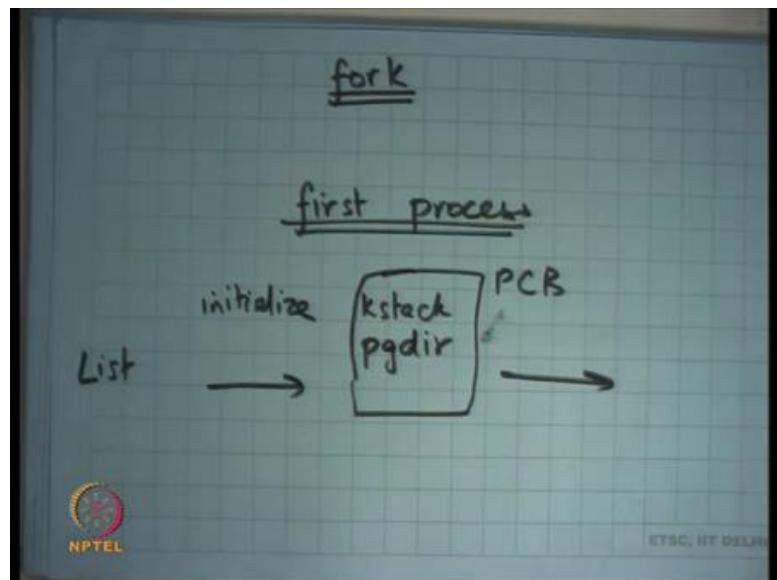
Another interesting piece, another interesting thing how does how does the computer keep time, how does it know how much time it has passed you know, how does it how does it keep track of you know how does it implement your clock?

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 context 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 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 k alloc 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 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

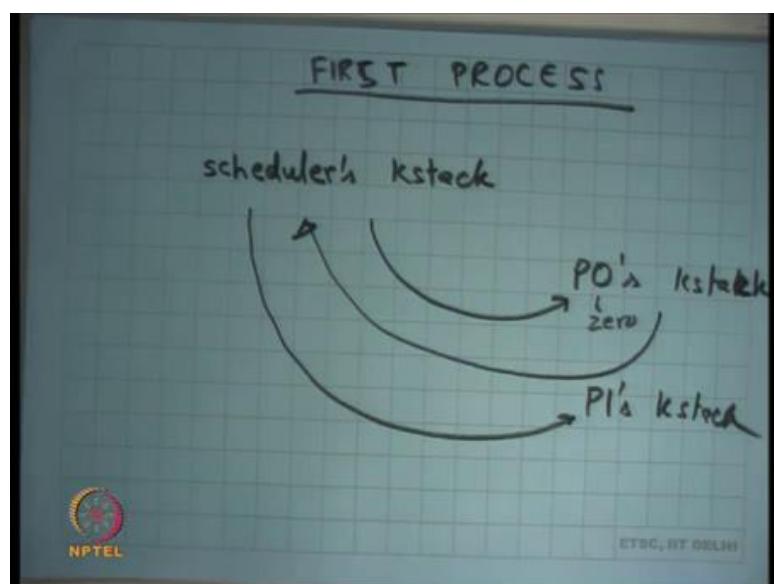
page faulted, and you are going to copy that particular page, and make two copies of it one for the child and one for the parent.

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 P 0's, let say the init process of P 0.

So, it is going to initialize P 0's kstack and its going to switch. So, this is 0 right, P 0's kstack, and it is going to switch from schedulers kstack to the P 0's kstack. The P 0 will now get to run, and P 0 may now let us say fork new processes. So, a new P 1 gets created, and so new P 1s kstack gets created right. And at some point, P 0 is going to yield either voluntarily by calling the yields is call or involuntarily because of a timer interrupt. In either case, P 0'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 P 1'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)

The screenshot shows a debugger interface with assembly code and a handwritten note. The assembly code is as follows:

```
2707 .globl swtch
2708 swtch:
2709     movl 4(%esp), %eax
2710     movl 8(%esp), %edx
2711
2712     # Save old callee-save registers
2713     pushl %ebp
2714     pushl %ebx
2715     pushl %esi
2716     pushl %edi
2717
2718     # Switch stacks
2719     movl %esp, (%eax)
2720     movl %edx, %esp
2721
2722     # Load new callee-save registers
2723     popl %edi
2724     popl %esi
2725     popl %ebx
2726     popl %ebp
2727     ret
2728
2729
2730
2731
2732
2733
2734
```

A handwritten note on the right side of the screen says "old context". Below it is a diagram of a stack labeled "old kstack". The stack has four entries: %ebp, %ebx, %esi, and %edi. An arrow points from the label "old" to the top of the stack. Another arrow points from the label "old context" to the same position. Below the stack diagram is the text "*old context = esp". To the right of the stack diagram is the text "esp = newcontext".

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 P 0 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)

context switch

save current into old

load from new

```
void swtch(struct context **old, struct context *new);  
    save current register context in old  
    and then load register context from new.  
asm  
    .globl swtch  
    .text:  
    .intel_syntax noprefix  
    movl 4(%esp), %eax  
    movl 8(%esp), %edx  
    # Save old callee-save registers  
    pushl %ebp  
    pushl %ebx  
    pushl %esi  
    pushl %edi  
    .text  
NPTEL  
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.

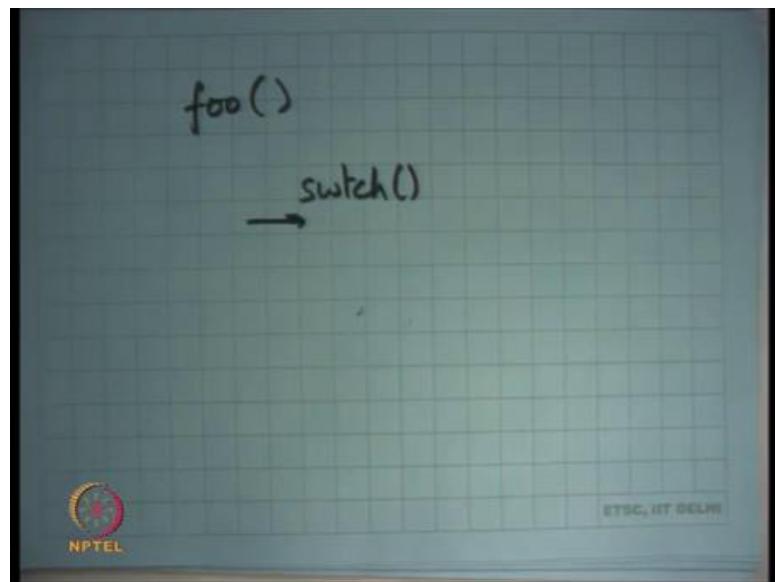
And then whatever you did here you do the opposite of here which is pop all the registers. So, because the old context, the new context must be assumed to be in the same structure as your current also. Assuming that that process was also switched out in the same way which means it must have pushed all 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 invariant 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?

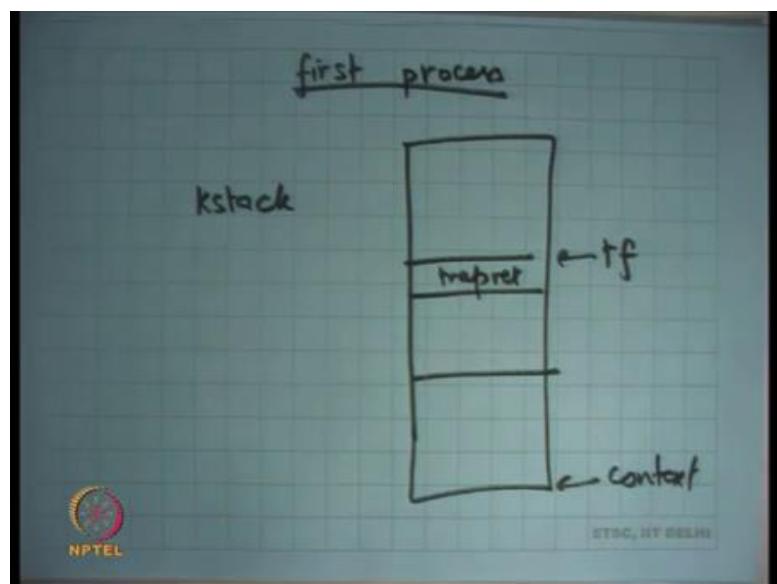
Student: The switch.

By the return instruction.

Student: Return instruction.

Right. So, the return instruction is the one that changes 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)

```
2100 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2101
2102 // Per-process state
2103 struct proc {
2104     uint sz;                      // Size of process memory (bytes)
2105     pde_t* pgdir;                 // Page table
2106     char *kstack;                 // Bottom of kernel stack for this process
2107     enum procstate state;        // Process state
2108     volatile int pid;            // Process ID
2109     struct proc *parent;         // Parent process
2110     struct trapframe *tf;        // Trap frame for current syscall
2111     struct context *context;     // swtch() here to run process
2112     void *chan;                  // If non-zero, sleeping on chan
2113     int killed;                  // If non-zero, have been killed
2114     struct file *ofile[NFILE];   // Open files
2115     struct inode *cwd;           // Current directory
2116     char name[16];               // Process name (debugging)
2117 };
2118
2119 // Process memory is laid out contiguously, low addresses first:
2120 //   text
2121 //   original data and bss
2122 //   fixed-size stack
2123 //   expandable heap
2124
2125
2126 NPTEL
2127
```

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)

```
3000 #include "mmu.h"
3001
3002 # vectors.S sends all traps here.
3003 .globl _lptraps
3004 _lptraps:
3005 # Build trap.frame.
3006 pushl %ds
3007 pushl %es
3008 pushl %fs
3009 pushl %gs
3010 pushl %
3011
3012 # Set up data and per-cpu segments.
3013 movw $SEG_KUDATA<<3), %ax
3014 movw %ax, %ds
3015 movw %ax, %es
3016 movw $SEG_KCPU<<3), %ax
3017 movw %ax, %fs
3018 movw %ax, %gs
3019
3020 # Call trap(tf), where tf=%esp
3021 pushl %esp
3022 call trap
3023 addl $4, %esp
3024
3025 # Return falls through to trapret...
3026 .globl trapret
3027 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 k data and seg u data 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 off 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.

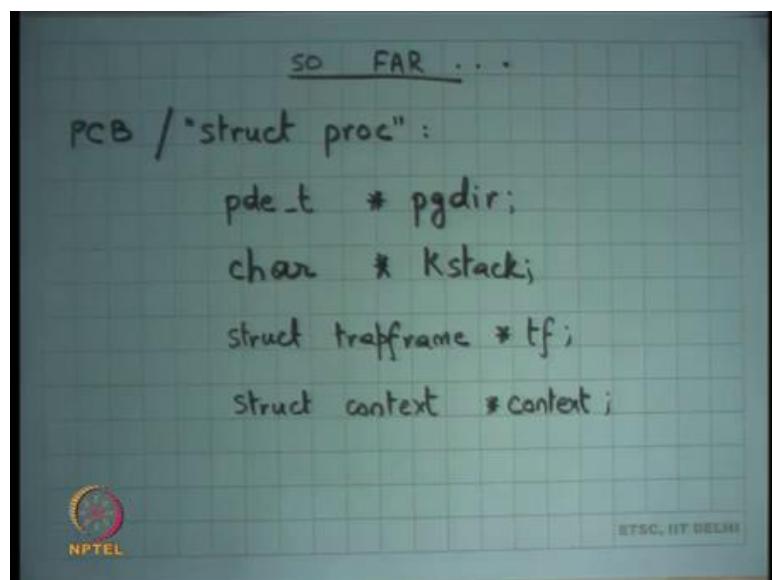
So, let us stop here, and we are going to discuss this more the next time.

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, 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 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).

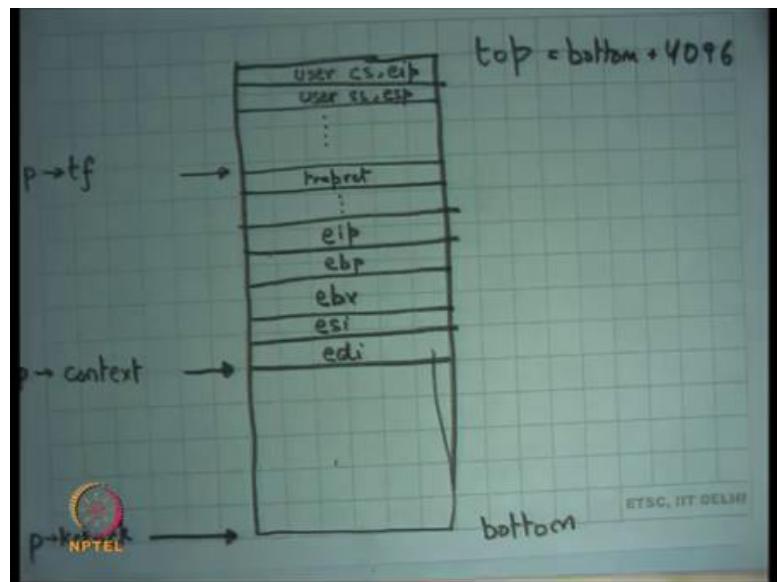
So, the CLI instruction that is the only way to modify the bit of the EFLAGS is the privilege instruction. And, so if the user ever executes that instruction what will happen? An exception will get thrown and the kernels exception handler will get called and the external exception handler may just kill 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 kalloced 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 know call some function which will have their own local variables and return address etcetera.

And, finally, somewhere here you will call switch right. So, 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 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)



So, today I am going to talk about 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_size);
    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 allocproc. What is allocproc going to do? It is going to allocate a PCB.

It is going to allocate a PCB and it is going to let us see what exactly allocproc 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 allocproc 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;
2226 }
```

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)



```
1  return 0;
2 }
3 sp = p->kstack + KSTACKSIZE;
4
5 // Leave room for trap frame.
6 sp -= sizeof *p->tf;
7 p->tf = (struct trapframe*)sp;
8
9 // Set up new context to start executing at forkret,
10 // which returns to trapret.
11 sp -= 4;
12 *(uint*)sp = (uint)trapret;
13
14 sp -= sizeof *p->context;
15 p->context = (struct context*)sp;
16 memset(p->context, 0, sizeof *p->context);
17 p->context->eip = (uint)forkret;
18
19 return p;
20 }
21
22
23
24 }
```

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 4 is equal to `tf` 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 `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)

```
2080 extern struct proc *proc asm("%gs:4"); // cpus[cpunum()].p
2081
2082
2083 // Saved registers for kernel context switches.
2084 // Don't need to save all the segment registers (%cs, etc),
2085 // because they are constant across kernel contexts.
2086 // Don't need to save %eax, %ecx, %edx, because the
2087 // x86 convention is that the caller has saved them.
2088 // Contexts are stored at the bottom of the stack they
2089 // describe; the stack pointer is the address of the context.
2090 // The layout of the context matches the layout of the stack in
2091 // at the "Switch stacks" comment. Switch doesn't save eip explicitly
2092 // but it is on the stack and allocproc() manipulates it.
2093 struct context {
2094     uint edi;
2095     uint esi;
2096     uint ebx;
2097     uint ebp;
2098     uint eip;
2099 }
```

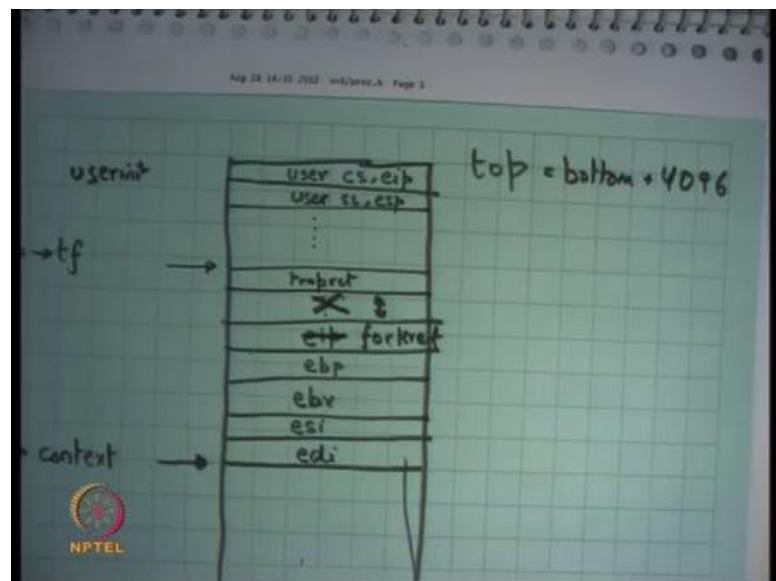
Let us look at their context once again refer sheet 20. This is the declaration for struct context and struct context has this five fields which basically look they are integer fields and they have been named on the registers that register values or register names edi, esi, ebx, ebp and eip alright.

Notice that 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_initcode_size);
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->cr3 = namei("/");
p->NPTEstate = RUNNABLE;
```

So, I have called allocproc here right and then I set up some global variable called initproc 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)

```
56
57 p = allocproc();
58 initproc = p;
59 if((p->pgdir = setupkvm()) == 0)
60     panic("userinit: out of memory?");
61 inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_end);
62 p->sz = PGSIZE;
63 memset(p->tf, 0, sizeof(*p->tf));
64 p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
65 p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
66 p->tf->es = p->tf->ds;
67 p->tf->ss = p->tf->ds;
68 p->tf->eflags = FL_IF;
69 p->tf->esp = PGSIZE;
70 p->tf->eip = 0; // beginning of initcode.S
71
72 safestrcpy(p->name, "initcode", sizeof(p->name));
73 p->cwd = namei("/");
74
75 p->state = RUNNABLE;
76
77
```

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

7700 # Initial process execs /init.
7701
7702 #include "syscall.h"
7703 #include "traps.h"
7704
7705
7706 # exec(init, argv)
7707 .globl start
7708 start:
7709     pushl $argv
7710     pushl $init
7711     pushl $0 // where caller pc would be
7712     movl $$SYS_exec, %eax
7713     int $T_SYSCALL
7714
7715 # for(;;) exit();
7716 exit:
7717     movl $$SYS_exit, %eax
7718     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?

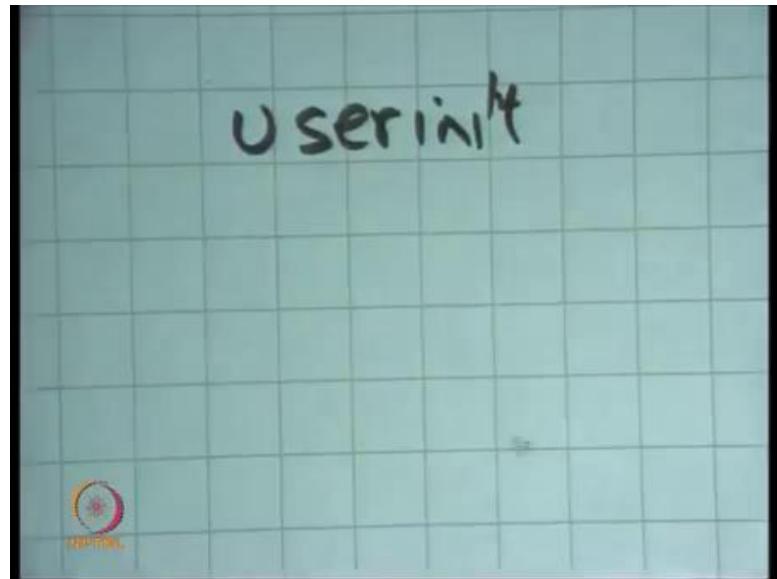
Student: (Refer Time: 26:40) any where is this files store? Actually, we do not have any file right now right so.

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)



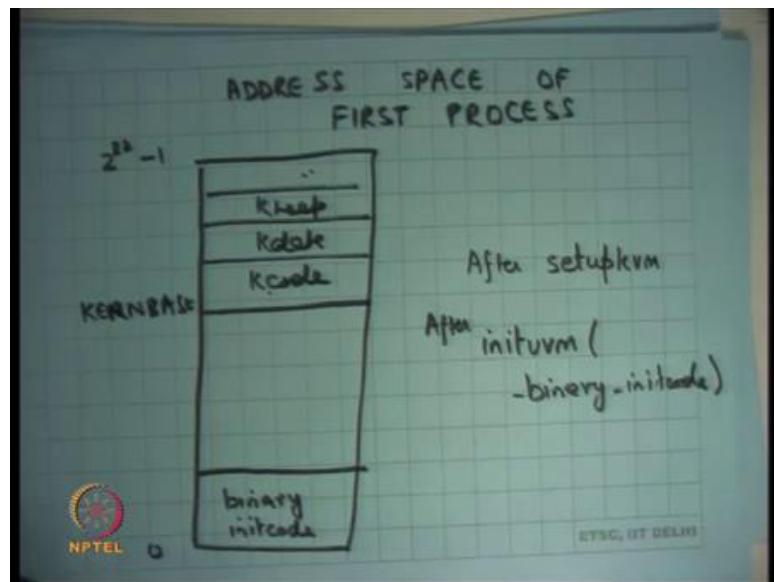
So, we are talking about userinit that is going to create the first process right. So, what is going to do we going to allocate a new PCB it is going to initialize the new page directory ok. And, it is going to paste the contents of the first program that it wants to run into this page directory into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

```
2256
2257     p = allocproc();
2258     initproc = p;
2259     if((p->pgdir = setupkvm()) == 0)
2260         panic("userinit: out of memory?");
2261     inituvm(p->pgdir, _binary_initcode_start, (int)_binary_
2262     p->sz = PGSIZE;
2263     memset(p->tf, 0, sizeof(*p->tf));
2264     p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2265     p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2266     p->tf->es = p->tf->ds;
2267     p->tf->ss = p->tf->ds;
2268     p->tf->eflags = FL_IF;
2269     p->tf->esp = PGSIZE;
2270     p->tf->eip = 0; // beginning of initcode.S
2271
2272     safestrcpy(p->name, "initcode", sizeof(p->name));
2273     p->cwd = namei("/");
2274
2275     p->state = RUNNABLE;
2276 }
2277 }
```

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.

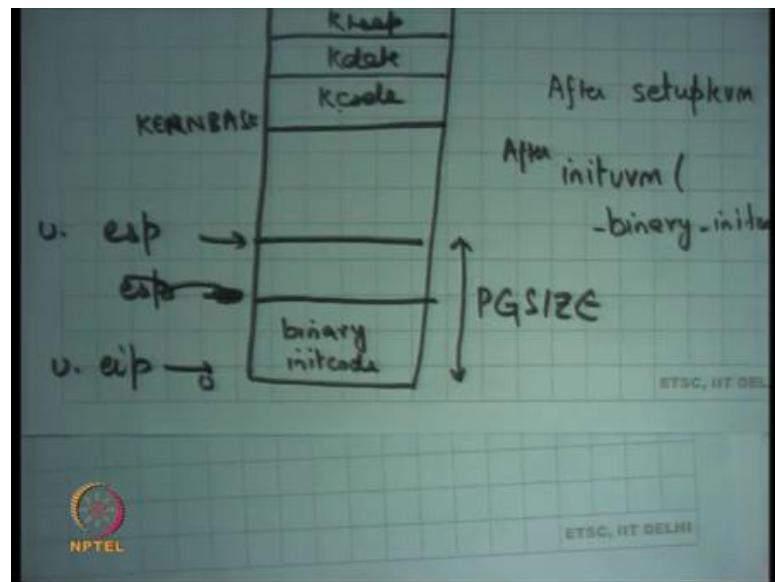
So, in this case it is going to start initializing the trapframe. So, it first 0s out all the contents of the trap trapframe. So, all the registers are 0 by default except cs is pointed to the user's code segment right. Similarly, ds are pointers to the users data segment. Why cannot we have done if 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 were 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)

```
2101
2102 // Per-process state
2103 struct proc {
2104     uint sz;                      // Size of process memory (bytes)
2105     pde_t* pgdir;                 // Page table
2106     char *kstack;                 // Bottom of kernel stack for this process
2107     enum proctype state;         // Process state
2108     volatile int pid;            // Process ID
2109     struct proc *parent;         // Parent process
2110     struct trapframe *tf;        // Trap frame for current syscall
2111     struct context *context;     // swtch() here to run process
2112     void *chan;                  // If non-zero, sleeping on chan
2113     int killed;                  // If non-zero, have been killed
2114     struct file *ofile[NFILE];   // Open files
2115     struct inode * cwd;          // Current directory
2116     char name[16];               // Process name (debugging)
2117 };
2118
2119 // Process memory is laid out contiguously, low addresses first
2120 // text
2121 // original data and bss
2122 // 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

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

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 proctype 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[NFILE];   // Open files
    struct inode * cwd;          // Current directory
    char name[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 inherits 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?

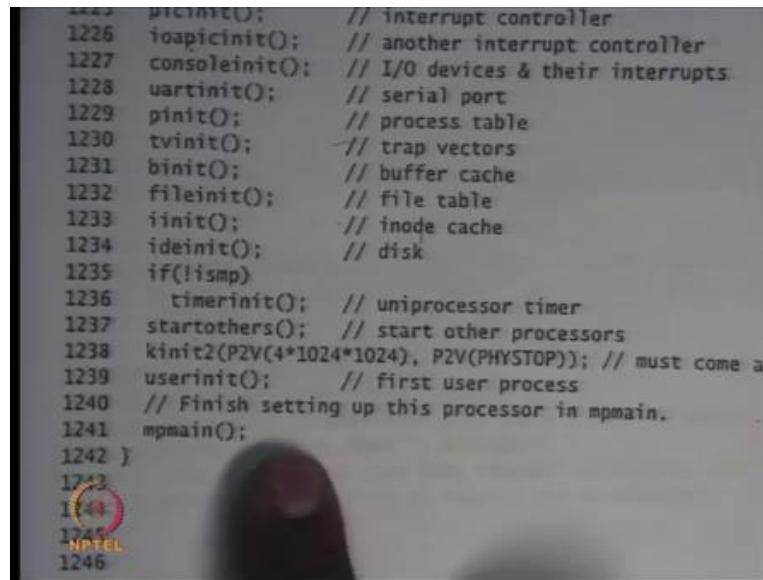
Why the type volatile? What does volatile mean I am going to discuss that later. Let us just know let just skip that for now want to discuss that later.

(Refer Slide Time: 41:08)

```
1216 int
1217 main(void)
1218 {
1219     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1220     kvmalloc(); // kernel page table
1221     mpinit(); // collect info about this machine
1222     lapicinit();
1223     seginit(); // set up segments
1224     sprintf("\ncpu%d: starting xv6\n\n", cpu->id);
1225     picinit(); // interrupt controller
1226     ioapicinit(); // another interrupt controller
1227     consoleinit(); // I/O devices & their interrupts
1228     uartinit(); // serial port
1229     pinit(); // process table
1230     tvinit(); // trap vectors
1231     binit(); // buffer cache
1232     fileinit(); // file table
1233     init(); // inode cache
1234     ideinit(); // disk
1235     if(!ismp)
1236         timerinit(); // uniprocessor timer
1237         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)



```
1225 picinit();      // interrupt controller
1226 ioapicinit();   // another interrupt controller
1227 consoleinit();  // I/O devices & their interrupts
1228 uartinit();     // serial port
1229 pinit();        // process table
1230 tvinit();       // trap vectors
1231 binit();        // buffer cache
1232 fileinit();    // file table
1233 iinit();        // inode cache
1234 ideinit();     // disk
1235 if(!ismp)
1236     timerinit(); // uniprocessor timer
1237 startothers();  // start other processors
1238 kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after
1239 userinit();     // first user process
1240 // Finish setting up this processor in mpmain.
1241 mpmain();
1242 }
1243
1244
1245
1246
```

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)

```
1256     lapicinit();
1257     mpmain();
1258 }
1259
1260 // Common CPU setup code.
1261 static void
1262 mpmain(void)
1263 {
1264     cprintf("cpu%d: starting\n", cpu->id);
1265     idtinit();           // load idt register
1266     xchg(&cpu->started, 1); // tell startothers() we're up
1267     scheduler();         // start running processes
1268 }
1269
1270 pde_t entrypgdir[]; // For entry.S
1271
1272 // Start the non-boot (AP) processors.
1273 static void
1274 startothers(void)
1275 {
1276     extern uchar _binary_entryother_start[], _binary_entryother_end;
1277     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)

```
3 // Enable interrupts on this processor.
4 sti();
5
6 // Loop over process table looking for process to run.
7 acquire(&ptable.lock);
8 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
9     if(p->state != RUNNABLE)
10        continue;
11
12     // Switch to chosen process. It is the process's job
13     // to release ptable.lock and then reacquire it
14     // before jumping back to us.
15     proc = p;
16     switchuvvm(p);
17     p->state = RUNNING;
18     swtch(&cpu->scheduler, proc->context);
19     switchkvm();
20
21 // Process is done running for now.
22 // It should have changed its p->state before coming back.
23 proc = 0;
24 }
```

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 have 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 to 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 be 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 the
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)

```
7807 char *argv[] = { "sh", 0 };
7808
7809 int
7810 main(void)
7811 {
7812     int pid, wpid;
7813
7814     if(open("console", O_RDWR) < 0){
7815         mknod("console", 1, 1);
7816         open("console", O_RDWR);
7817     }
7818     dup(0); // stdout
7819     dup(0); // stderr
7820
7821     for(;;){
7822         printf(1, "init: starting sh\n");
7823         pid = fork();
7824         if(pid < 0){
7825             printf(1, "init: fork failed\n");
7826             exit();
7827         }
7828         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.c and init.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)

```
7814     if(open("console", O_RDWR) < 0){
7815         mknod("console", 1, 1);
7816         open("console", O_RDWR);
7817     }
7818     dup(0); // stdout
7819     dup(0); // stderr
7820
7821     for(;;){
7822         printf(1, "init: starting sh\n");
7823         pid = fork();
7824         if(pid < 0){
7825             printf(1, "init: fork failed\n");
7826             exit();
7827         }
7828         if(pid == 0){
7829             exec("sh", argv);
7830             printf(1, "init: exec sh failed\n");
7831             exit();
7832         }
7833         while((wpid=wait()) >= 0 && wpid != pid)
7834             printf(1, "zombie!\n");
7835     }
```

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.

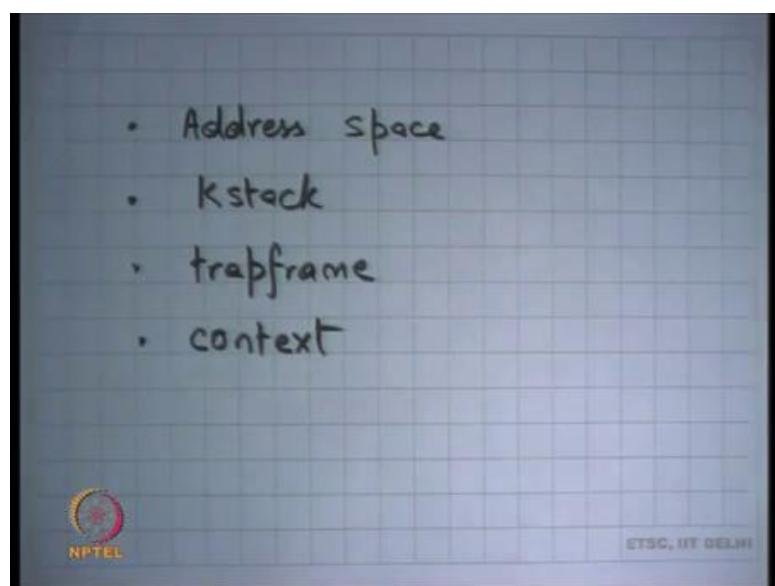
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 20
Handling User Pointers, Concurrency

Welcome to Operating Systems, lecture 20. So, we have been looking at how a process is implemented, a process has an address space in xv6 it also has a K stack. So, there is a stack for process which is a kernel stack for process. So, every process has a separate stack on the kernel and every process behaves as a thread inside the kernel right, because it is a shared address space inside the kernel for every process.

While the process is executing in the kernel side then there is something called a trap frame that gets pushed right at the entry time, right and that is the trap frame. The trap frame basically contains all the values of the user registers at the time of the trap and can be used to do different things. For example, it can be used to implement system call arguments and system call return values, right.

(Refer Slide Time 01:11)

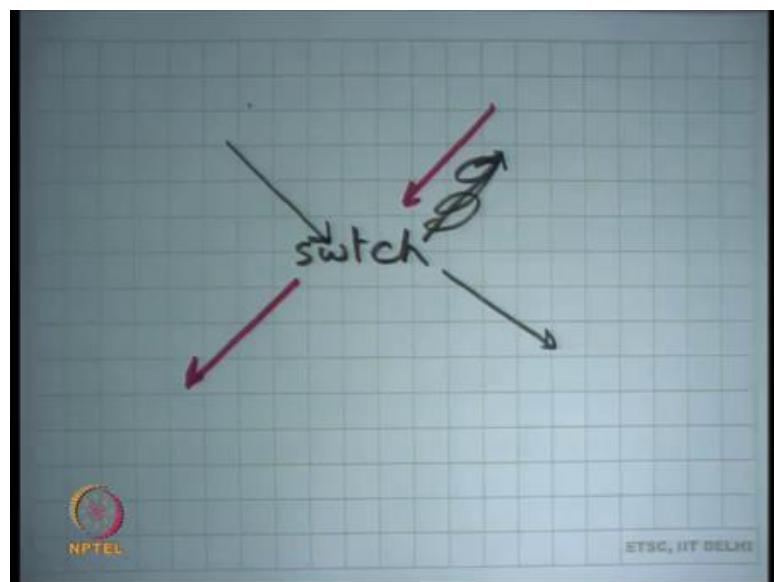


And, then we said there is this structure called context which is also stored as a stack. It is really at the bottom of the stack right, for the switched-out thread the context structure is at the bottom of the thread and of the stack. And, basically it is stores the values of the kernel registers at the time the switch was called, right; so, including the eip, right So,

when you switch to this thread, you are going to reload those registers and reload the eip by calling the return instruction and you continue from where you left, right..

So, basically there is some function which will call switch function and you know whoever calls the switch function is not going, the switch function is not going to return back to that function; switch functions going to return to some other function, right. So, the switch function was something very special.

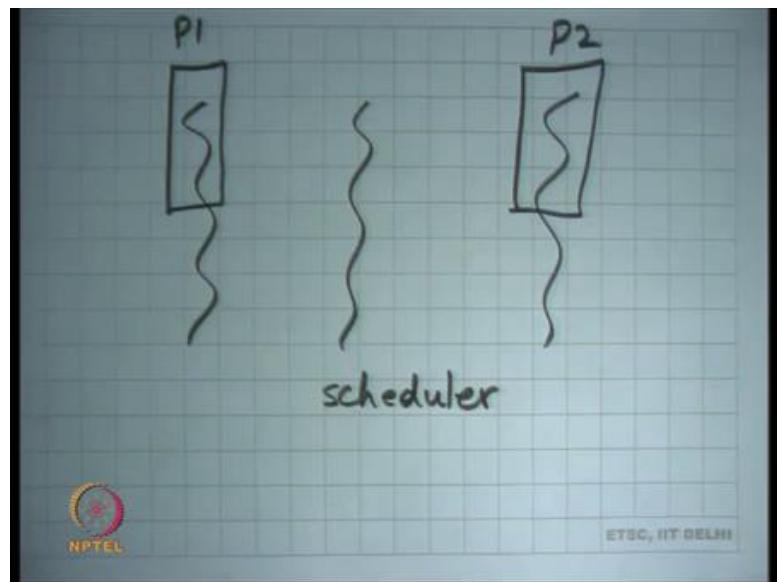
(Refer Slide Time 02:05)



And typically, the way it works is that let us say this is switch, right. So, this is one thread that calls it and, but return does not happen on that thread, return happens somewhere else, right. So, let me draw it like this, right. And, at some point later this thread is going to call switch again and returns going to happen here, alright.

So, this is this special function where you know the caller does not get to see the return alright, and that is how you implement the context switch, right. And we said, the context switch is basically the core of the kernel, because it does all the switching logic, everything in the kernel is either a process or a thread, right.

(Refer Slide Time 02:57)



So, if I were to say what is the you know if I will look at the kernel, then you have these multiple threads. Some of these threads are backed by address spaces; these are called processes, right. There could be other threads that are not backed by processes; they are just doing kernel functionality, right.

So, let us say there is some thread that just checking if everything is and has nothing to do with any process or anything of that sort. So, let me just be a kernel thread without any address space of its own, right. The only thing it has is the kernels address space, right. So, recall that any process has the kernel address space and a user address space. So, these rectangles there are showing the user address spaces and this one does not have a user address space, it is only a kernel thread.

So, once that example of a kernel thread is the scheduler thread right, alright. So, this is the thread for example that was started right at the beginning. So, when you booted the program booted the kernel you started a stack and you initialize the stack, and you started executing some piece of code on that, and in eventually you called the scheduler on that stack, right.

And so, when the scheduler gets to run, he is running on the stack and then what this scheduler will do is it will look at this array of processes and pick up one that is runnable. And, just switch using the switch function to that thread right, that thread is going to run and let us say that thread was the process.

So, then the process is going to run because it is going to do array trap rate and it is going to get back to user mode, that process is going to get to run for some time. It is possible that the process says, I will not I do not want to run anymore in which case it makes a system call called yield let us say, it makes a system call called yield internally also called switch, right.

So, yield will call switch to switch this scheduler, right and then the scheduler will again get to run, before it calls switch to switch scheduler it will change its state from running to runnable, right. Now, scheduler gets to run, and it picks up one of the runnable processes. It is possible that the same process gets picked again alright where it is possible. Whatever, the scheduling policy is if it is round robin let us say, then you know some other process will get to run if there were other processes in the system, right.

So, then; so, basically the chain of control is scheduler, P 2 scheduler, P 1 scheduler, P 2 and so on, right whatever. So, you switch the stack from P 2 K stack to the scheduler scale stack and then you switch from schedulers K start to P 1 K stack and so on.

Student: Sir.

If the switch is happening on the yield system call are the interrupts disabled well as it is you know it is up to the operating system, whether it wants to disable interrupt from a system call or not, right. In general, an operating system will not disable interrupts on a system call, alright. So, yield is just like any other system call, so it is not going to disable interrupt on at that point.

Student: Sir.

But, if there is an external interrupt while the yield was executing, then the handler of that external interrupt will execute with interrupt disable, right. So, we said that all external interrupt handlers must execute with interrupt disable to have a bound on the K stack, ok.

Student: Sir; so, you are.

Ok.

Student: Another context.

Alright. So, the other way for a context switch to happen is that you know the previous example was P 2 call a system call called yield. But it is possible that P 2 never call the system call called yield which is you know usually when you write your program you never call yield really, right. You just expect the program to run and still be friendly with all the other programs in the system.

So, the way here happens is let us say you are executing in P 2 a timer interrupt occurs, the timer interrupt handler gets called. The timer interrupt handler internally will probably call the same function that the yield system call would have called, right and which will internally called a switch function and it will do the exact same thing as a yield system call, alright.

The difference; the only difference between an external interrupt and a system call is that the external interrupts execute the handler of the; external interrupts executes with interrupted disabled and the system call may or may not execute with interrupts disabled, ok.

It is possible that while you were executing the system call handler and the system call could be any system call including the yield system call, a timer interrupt occurs, or any other interrupt occurs. And so, what will happen is you will continue pushing the trap frame on the kernel stack. So, the kernel stack would have two trap frames right and that is perfectly possible. Could the kernel stack have to context?

Student: No.

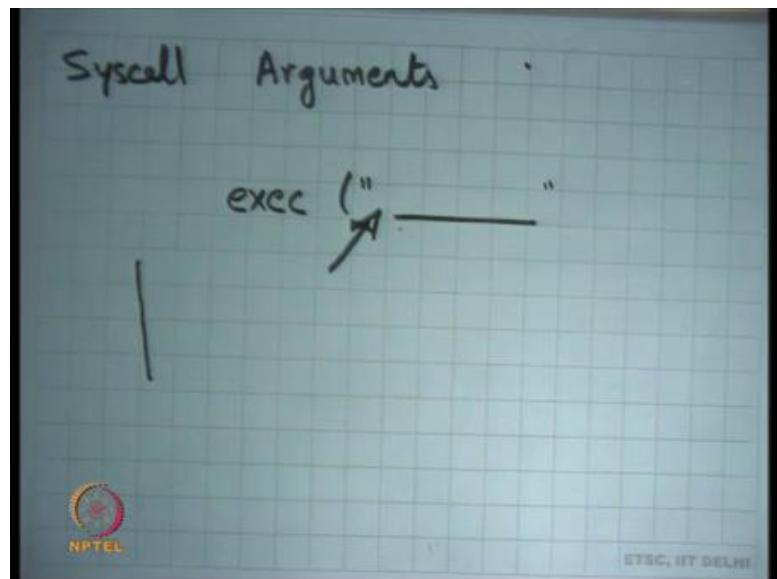
No; it cannot have two contexts because by the time you push the context, you have already switched.

Student: Yes sir.

Alright, and you also ensure which we are going to see later on that while you are pushing the context, the interrupts are disabled. So, while you know it is not possible that you push two registers of the context and then an interrupt occurs. And so, you know half the context is there and you know another context this is not possible, because the kernel just carefully disables interrupts while it is context switching. And, as soon as the context switch it can just you know re-enable interrupts or revert to the previous value of

the interrupt flag, whether it was enable or disable depends on you know how you can there, alright ok

(Refer Slide Time 08:23)



So, good. Now, today I am going to talk about how you handle you so; so, I said arguments are passed to syscall, right. So, syscall arguments, right. And one of the big reasons we had this organization of an OS where there is a kernel and the users sharing the same address space is because user and kernel can communicate very fast, they can just extend a pointer and the user can dereference a pointer because we are in the same address space, right. So, that was a big deal.

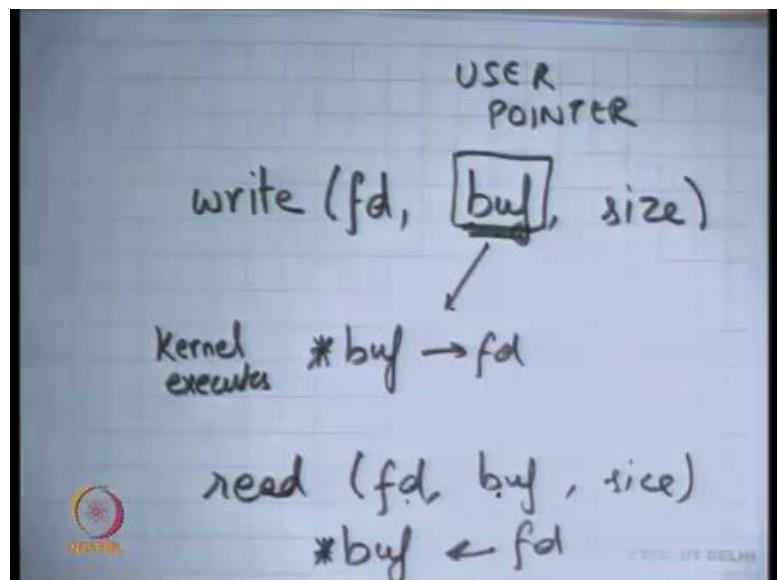
And, what we said was let us say I have a system call called exec and it takes a string, right. Now, a string can be of arbitrary length. So, clearly a string cannot be passed into registers or even you know on passing the string on the stack may not be the right thing, because you know stack it is you know, right. So, you could pass it potentially on the stack, but let us say you know you cannot pass it on the registers.

So, one way; so, the way it is done is you pass the pointer; so, you, what the user does is it initializes the string somewhere. The string could be either on the user stack or it could be in the users heap or it can be in the users data section; it does not matter, it is in the users address space that is all I care about, right.

So, the string is in the user address space and what I gave to the kernel is a pointer to this string and address right. And, the kernel guesses address and the kernel is supposed to dereference this pointer to get the value of the string, ok.

Now, what can happen? Well, what can happen is if the user is not trusted which is which it is not in general, right. What can happen is the user can give a pointer, which is let us say a pointer in the kernel address space, right and so, it can ask the user can ask the kernel to dereference that pointer for itself, alright.

(Refer Slide Time 10:31)



So, let me give you a concrete example let us say I wanted to call the write system call, alright. So, write takes a file descriptor and the buffer and the size, let us say ok. So, once again the buffer is a string, I basically communicated using a pointer. So, what do I tell kernel; I say, here is the address and just take you know size bite from this address and put it in the file, alright?

Now, if I was a malicious user what I can do is this pointer can be a pointer in the kernel address space right and the kernel is just going to just dereference it, right. So, the kernel is dereferenced it even if the pointer was in the kernel address space because the kernel was executing in privileged mode. It will be able to dereference it and you will be able to read all those values, and you will now write all those values into the file, right. So, what can the user do, it can actually read the entire kernel, right, yes question.

Student: Processing, while kernel is dereferencing the address kernel essentially know that this and this is about (Refer Time: 11:35), that is why it is in kernel space. So, it will just create a faulting a case.

So, I am just saying that if the kernel was to just deference buf without doing any checks, then there is a problem. Do you agree?

Student: Yes.

Yes, so, if the kernel executes star buf and puts it into file descriptor fd and the user can potentially read the entire kernel, right. It just needs to craft his point in such a way that you know it looks at different addresses of the kernel and I can look at the entire code and the entire date of the kernel. I can look at the date of other processes, it because after all the other processes are mapped in the kernel space also including myself and so, I can look at the data of other processes, right.

Similarly, in fact, so, this is just you know this is a way of reading the kernel, I could actually crash the kernel. So, in fact, the read system call can cause can allow a process to write any arbitrary data to any arbitrary pointer.

So, here what is going to happen, kernel is going to execute star buf gets whatever the contents are at the current offset, right. And so, I can what the kernel what the user can ask the kernel to do it can just say oh change this memory location to something else alright, and it can ask him to do whatever he likes for that matter. So, this is a security flaw, right.

So, these are this is so, what this buf or any other pointer that is passed as an argument to a system call by the semantics is called a user pointer, ok. A user pointer is supposed to be untrusted in the kernel; a kernel should never trust a user pointer. It should always check the validity of a user pointer before ever dereferencing it. So, what can what kind of checks can the can kernel make? Well, one thing is buf should never be above kern base, buf should always be a value below kern base, is that enough.

Student: (Refer Time: 14:01), page faults.

Right so, it should be an address that is actually mapped in my address space. It because if it is not mapped then what I can do, what the user can do is ask the kernel to deference

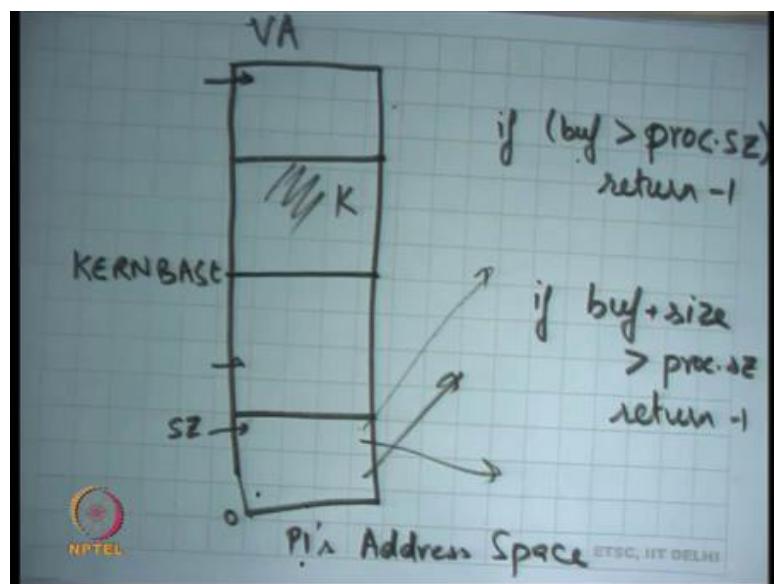
a location which will cause a page fault, right. And, the kernel is, may not be capable of handling page faults while it is executing, right. If a kernel maybe may be capable of handling page faults while the user is executing. But the kernel may not be able to handle its own page faults, right or kernel does not expect itself to have page fault, right and so, that can cause problems; right.

So, basically the two things; you have to check; you have to check that the address is mapped and that the address is mapped with user privileges not with kernel privileges. In other words, the user himself should be able to read or write to that address. In other words, user asking the kernel to read or write to that address, but the kernel should only do that if the user himself is allowed to read or write to that address, ok.

And so, so, that may involve for example, checking that you know in our xv6 organization it may involve checking that buf is less than kern base 2 GB and buf when you walk the table direct page directory page table of that process then buf is mapped. And, buf is mapped in user privileges and in you know if it is read system call then it should be writable, right.

So, if our processes doing different privileges for different areas; so, one page is writable another page is readable, then you should only be able to call the read system call on a writable page not on a readable page. So, it can do all these checks ok, alright.

(Refer Slide Time 16:00)



So, in fact, if you look at xv6 let us say this is the address space ok, this is kern base, and this is kern base. And oh, there is some kernel data here, I am just drawing, call it K and, what xv6 organizes it is address spaces that all P 1 space should be from 0 to some size, right. So, that is you know xv6 takes a simple approach, it says that the users address space the processes address space should be limited between 0 and size, ok.

Of course, in the physical memory it can be all scattered, it does not matter, but in the virtual address space. So, this is of course, virtual address space, it should be from 0 to size. On Linux for example, that is not true you could you know you could just have you know a segment here and a segment here anything is possible.

Paging gives you full flexibility, you can have would really dis-contiguous mappings, but Xv6 decides to use a contiguous mapping for it is processes perfectly ok, alright. And so, what does the Xv6 kernel need to check that any user pointer should lie between 0 and size right, before it dereferences it, right.

So, for example, if if I do a right fd buf size all in which do is, if buf is greater than size is proc let us say proc dot size. So, proc is you know the pcb structure of that process, then you know just say sorry I just return minus 1 for that particular system call alright, otherwise dereference for example, alright.

Actually, it is not so simple because you do not just have to check buf you should be checking really buf, buf + 1, buf + 2 all the way till buf + size, right, And so, question is how what is the minimum number of checks I need to make, is it to just check check buf? No, because I could have given buf here and buf + size could be here. And so, you know when I dereference buf + 10 or something then it can cause a page fault; so, that is not right is it to just check buf + size. So, instead of if buf is get instead of this if I just say if buf + size is greater than proc dot size return minus 1, otherwise just continue and start dereferencing, it is this correct?

Student: Size (Refer Time: 18:55).

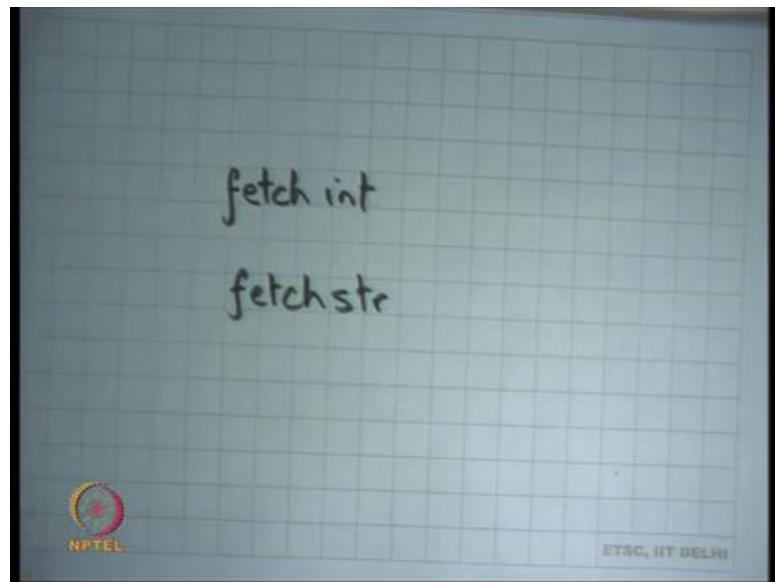
So.

Student: (Refer Time: 18:58), it may cycle back and then.

Right. So, assuming that sizes are unsigned integer. So firstly, you know just assume that the sizes and unsigned integer it is not a signed integer, is this correct. So, there is one answer which says it may cycle back actually. So, what do you what the programmer can do is give a buf here right and so, buf plus size will be somewhere here because it is a 32-bit number? So, it will just wrap around, and it will come here. So, you may be going to check oh buf plus size is less than size and you are going to start dereferencing it and that is an arrow.

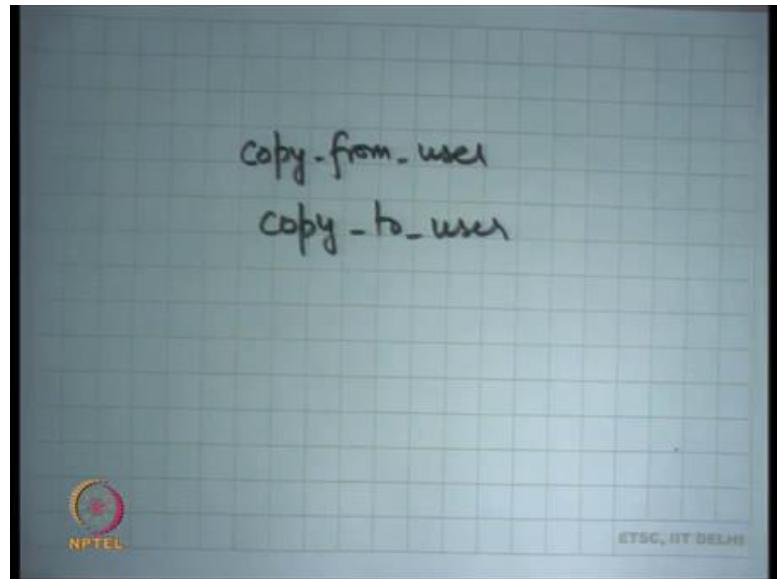
So, what do you need to check, you need to check both actually? So, it turns out that you need to check both you are done, right. So, you check buf, you check buf plus size and you are done right, because; that means, that both buf is in this area and buf plus size is in this area. So, the entire string must be in that area, alright ok.

(Refer Slide Time 19:57)



So, I will encourage you to look at these functions in Xv6 code called fetch int and fetch string, right. These are the functions that are doing these checks before dereferencing and use a pointer. These are functions that are designed to handle user pointers inside Xv6, right. So, the convention that the Xv6 programmer has followed is that any way I have to dereference a user pointer, I will not I will never do it directly I will do it using these special functions called fetch int and fetch string, right. So, that and these functions should do the appropriate checks, ok.

(Refer Slide Time 20:42)



Similarly, on Linux, where these functions called copy from user and copy to user, right; once again, these are special functions designed to handle user pointers, right. If you happen to miss; so, for example, you know kernels code is relatively large and it is possible and multiple people are contributing to a kernel for like you know Linux or any other operating system.

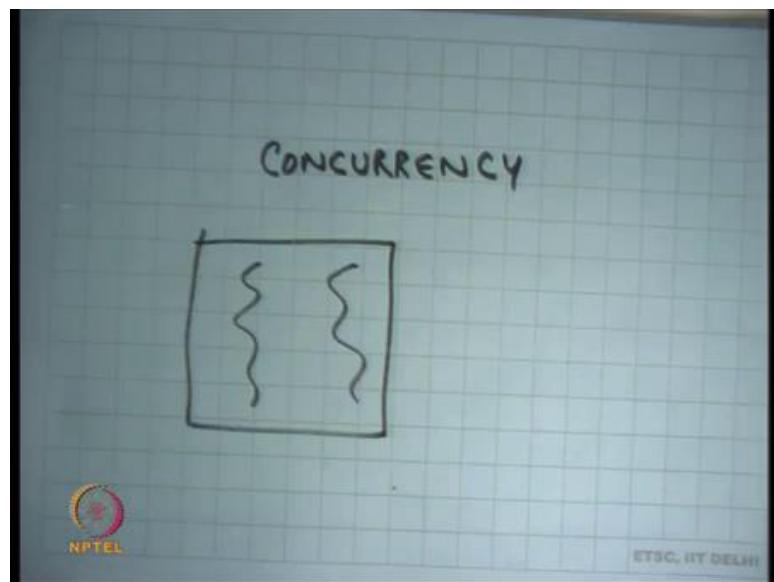
And so, it is quite possible that a programmer makes a mistake and one of the very common mistakes that were happening in the first you know 10 to 15 years of let us say Linux was that programmers were forgetting to actually use these functions to deference user pointer and they were just you know somewhere by mistake dereferencing the pointer directly. Because, you know that is that sort of you know it did not occur to them that this could be a user pointer, right.

So, because some of the some of the functions could be call if both on user pointers and on and on kernel pointers and those function will just dereferencing happily and that is you know they should have prevented that. So, these kinds of bugs are actually pretty common in the beginning and they were very easy targets for security exploits, right. Lot of work has happened since then and today will be very difficult to find such bugs in modern operating systems, right.

Your first; your second assignment on pintos I am ask you to implement these checks, right. So, when you are going to do system call handling and you are going to implement

the system calls. One of the things you will have to do to make sure that all your test pass is to make sure that even if the user gives you bad pointers you are handling them gracefully, right. You are not it does not cause your kernel to crash or behave an unexpected way, alright good ok.

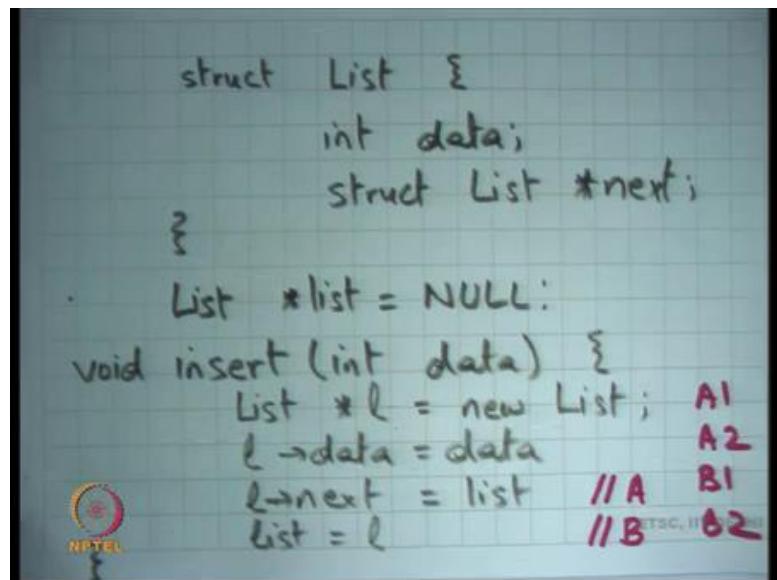
(Refer Slide Time 22:38)



So, now I am going to talk about another very interesting topic and perhaps the most practical and useful topic in this course called concurrency, right ok. So, what is concurrency? So, we said that you know every process is a thread. So, we already know that there is a notion of a thread and the idea is that threads execute in the same address space, right. Typically, when we write code, we assume that we are the only one running, right. So, let us take an example to illustrate this.

So, for example, on the you know this I am going to write some code that is taken from the ide device driver of Xv6 which is just a link list manipulation code.

(Refer Slide Time 23:23)

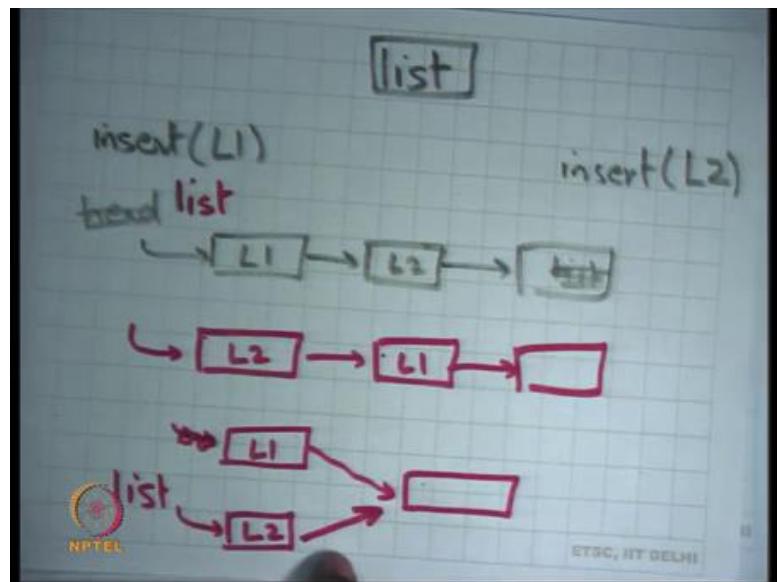


So, you know the device driver for the disk on in xv6 device has the structure called list where each element has a data and then there is a pointer. So, it is a link list and I am sure you are all familiar with link list, alright. And you know initially; so, I am just going to call it list instead struct list just assumed as a type def there. And so, let us start list is equal to null that is the initial configuration.

And then there is a function called insert that takes a data element integer right and does what just says, ok. So, let us say this is void insert and I want to write the code for this. The code is very simple just says list star l this is local variable is equal to new list right, you know I am I am using new instead of malloc. But, basically allocate some space of the heap to create a node of the list and say l dot data is equal to data, right. And you say l dot next is equal to list alright, could you just append it in the beginning of the list right and just say list is equal to l that is your insert function.

So, whoever rise this will probably think that this is correct, right. In fact, it is correct under serial execution, but if two threads try to insert different data into the same list then what happens? So, let us see; so, we all understand this.

(Refer Slide Time 25:35)



So, let us say this was list originally one thread calls insert L1 and other thread calls insert L2, right; then what happens well we would expect that if there are two threads and they can execute concurrently. So, you can think of those threads as just you know bad by processes. So, both processes are made some system calls that want to write to the disk for example, right.

So, both of them are going to make the insert system call and what will happen is that either you will have something like L1, L2 and list right and let us say this will be the new list head, right. So, let me just call it list; this pen is not either this can happen or this can happen and both of these are acceptable, but what is the third thing that can happen that is not acceptable.

Student: Delay (Refer Time: 26:53).

Right. So, what will happen, if let us say this statement was statement A and let us say this statement was statement B, right. So, what happens if thread 1 executes A then thread 2 executes A right and then thread B1 executes b and then thread 2 executes B. So, the schedule is A1, A2, B1, B2.

Now, let us see what is going to happen. Well, I am going to say let us say; so, they are going to be two elements L1 and L2 and you have a list originally. And so, L1 is going to say I want to point to list. So, he is going to point to list then A2 is going to execute. So,

L2 is going to be thread 2 so, I want to point to list. So, he is going to point to list like this and now B1 is going to execute; so, it is going to say list is this right. And, now B2 is going to execute I am going to say he is going to say oh list of this and so, the ultimate structure I may have is something like this, right ok.

So, if you if you if the executions get interleaved instead of L1, L2 this, this was a valid output because it is a it is still a well formed list, this is also well formed list, but this is not a well formed list, right. So, everybody agrees that this can happen, ok.

So, what is the problem? Problem is that the programmer when he was writing this code. He assumed that this code is going to execute in isolation, but this code is not going to execute in isolation if your program supports multiple threads in the same address space, right. Kernel is one example of a program that supports multiple threads in the same address space. You could write your own program that has multiple threads either user level or kernel level does not matter in either case there are multiple threads in the same address space and so, this program becomes incorrect, alright. This kind of an error is called a race or a race condition, right.

So, the idea is that one of them is doing something I am in the middle of doing something and I am not finished doing it and somebody else comes and he starts spoiling my intermediate state and so, the end result is come something completely wrong, ok. One day to achieve this would have been that I somehow ensure that while one thread is in this code no other thread can enter this code right that is one way to ensure that, ok. With that if you could ensure that then you know the programmer can write his code just like you would write any other serial code and his code will still be correct, alright.

So, this is you know similar to anything else where you have computation on shared state. So, for example, you are you know you like typing your program in an editor like vi. And typically, as a matter of habit you will first do you know you will first save the file and then you will call the compiler like let us say gcc on that file and that is a matter of habit.

This is just synchronization in your own mind going on between one program vi and another program gcc, right. If you do not wait for the same message to appear and you try gcc apriory then bad things are going to happen, right. And you cannot even predict

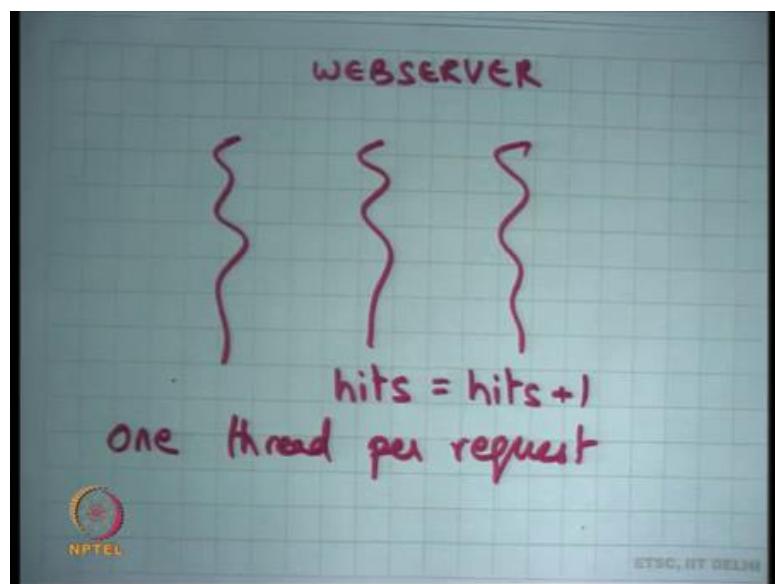
what kind of bad things can happen right which blocks were return, which blocks were not return completely out of your control race condition, ok.

Other examples well I mean even in physical world you know one road multiple cars they better follow some discipline otherwise they are going to collide, right. So, that is a race condition basically or a crossing, right. So, there are the one way to do one way you deal with road crossing is basically you have traffic lights, you say now you can go and now you can go and so on. So, you basically do round robin scheduling and that is how you basically ensure that when you know it is not like you get you know you collide with each other. So, you do some kind of synchronization.

Another example let us say classroom scheduling right. So, I am here teaching you a class, there was somebody else was teaching your class before me and there will be somebody else who will teach the class after me, and how our collisions prevented, how a bad things prevented from happening, this is the timetable that is put up on the board and you know we are all following the timetable, right.

So, these are all examples of shared state in this case a classroom and multiple threads you know we are all threads and you know why done we collide because we are doing some synchronization. Similarly, threads need to do some synchronization to make sure that they do not do these bad things, ok.

(Refer Slide Time 31:58)

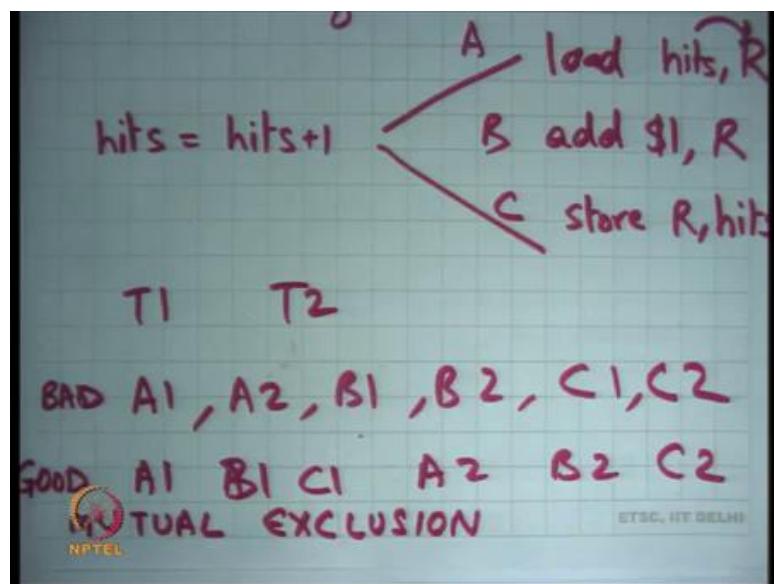


Let me take another example. Let us say I am a web server and I have multiple threads in the web server, right. You are running a website and your web server and what you have done is you have said that each thread is going to handle one request. So, one thread per request, right; so, whenever I type let us say www.iitd.ac.in you know my request will spawn a thread in the system and that thread will take my data, process it, get me the data you know get the content from it is local file system and serve it on the network and I see it on my browser.

So, if multiple threads are doing it simultaneously, multiple you know multiple that is how concurrency supported, in the same web server multiple people can access simultaneously, alright. So, let us say I have had this global variable called hits that was basically you know trying to understand what is how many hits do I get how many people actually visit my web page in a day. So, that is this variable hit and this this you know all you do is you say hits is equal to hit plus 1 for every request you get, right.

So, let us see what happens what happens if two threads try to exit, execute hits is equal to hits plus 1 simultaneously, alright ok.

(Refer Slide Time 33:26)



So, let us see hits is equal to hit plus 1 gets translated into let us say load hits into a register R right, and then it says add 1 to register R alright and then say store R to hits. I am using load in store, but you know we know that on Xv6 it is move instruction which does both load and store, alright.

So, let us say this is the code that gets generated in assembly level for the c statement called hits is equal to hits plus 1. And, once again what can happen is let us say this is statement A, B and C and there are two threads: T1 and T2. If schedule is A1, A2, B1, B2, C 1, C 2, then what is the final value of hits?

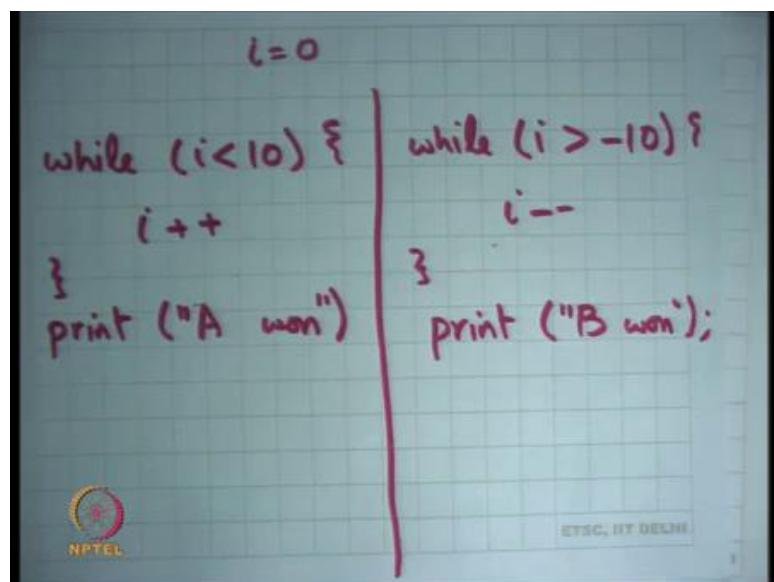
Student: Hits plus 1.

Hits plus 1, alright; so, will happen is both threads will load hits into their private registers both of them will. So, let us say initially hits was 0. So, they load the value 0 into the private register [vocalized- noise] each side has the private register, right. So, each thread loads a value 0 then each of them increments it; so, you get 1 in both registers. Now, both of the registers write 1 to the final output, right. So, what you get is hits is equal to hits plus 1, on the other hand if you had A1 A1 B1 C 1 A2 B2 C 2 then the value would have been?

Student: Hits plus 2.

Hits plus 2; hits is equal to hits plus 2, right. So, the final value is scheduled independent firstly so, it is a race condition again. And, you some of these values are incorrect, now what the correct value probably should have been hits is equal hits plus 2 right was the 2 requests that are happened, ok.

(Refer Slide Time 35:44)



So, another example of a race-race condition, alright and let me just give you one more example. So, let us say there is you know the two threads and one of them is saying while ($i < 10$) $i++$ right and say is you know when it completes it says oh print A1, right. And let us say there is another thread which says while ($i > -10$), $i--$ and print B1.

So, if this my program and then these are two threads one of them is trying to implement I, another thread is trying to determine i. What will be the final output and you know whoever finishes whoever reaches 10 or -10 first wins you know it wins happily that I am one and so. So, one of the possible outputs A1 B1, another output is B1 A1. So, whoever wins first he has 1 right, you cannot really you know there is no way a.

So, the semantics of threads do not tell you who is going to win they just say anybody can win. Is it even guaranteed that the program will ever terminate? Now, actually this program may not terminate right really right in theory, right. There is a nonzero probability that this program just keeps shuttling between values, but never reaches neither reaches -10 nor reaches +10 right. These are you know bad situations; programmer does not want this kind of uncertainty in general.

So, what is the problem, what is the solution, what is the possible solution? Well, one solution is do not do anything assuming that this kind of situation is acceptable, right. So, the example I gave you an example of web server hits. You may say ok, what is the probability that I will miss an update to hits.

So, I said that in general if you these were three instructions to update hits, if they execute like this everything is ok; if they execute like this, everything is ok; if they execute like this then there is a problem. What is the probability that they execute like this? Relatively small because all what you need is that that thread should be executing in exactly in the middle of these three instructions.

The probability is very small, and you may say oh I do not really care about you know the exact number of hits my website gets you know I just want to get a ballpark figure. So, you know I do not care if it is 1; if it was supposed to be 1 million if I even if I get a number which is one million minus one thousand, I do not care.

So, in this case you will just ignore it and you say let it be, ok. So, that is a valid response in some situations, but not in every situation. For example, the linked list example now

that is completely an invalid response you have corrupted your data structure and so, everything in future will actually be wrong, alright

The other the other responses try to avoid sharing right and there two threads why cannot you just duplicate state, right. So, you want to have hits, you want to count the number of hits at the end of the day just have two variables; one for this thread and other for this thread and let them update their private variables. And, at the end of the day just some of those variables and get your answer right that is another valid response works in some cases, right. And, you should try to do that as much as possible avoid sharing if you avoid sharing you avoid these bugs, these problems.

But what these are both of these are not general solutions. The general solution is to avoid bad interleaving, right.

(Refer Slide Time 39:21)



So, I want to avoid bad interleavings, So, what a bad interleavings in this case, well this is bad alright, and this is good ok. So, I have some notion of what is a bad interleavings and what is a good interleaving and I will avoid the bad interleavings and allow the good interleavings. In general, when we write code, we assume see an execution and so, bad interleavings are usually interleavings that involve overlaps right and good interleavings are interleavings that involve mutual exclusion, right.

So, in general good interleavings are usually executions that involve mutual exclusion. So, while I am running here you are not going to run here. So, you are going to run after I have run. It does not matter the order in which I run. So, even this is a mutually exclusive execution, and this is also mutually exclusive execution, this is not a mutually exclusive execution right. So, mutual exclusion is usually a good way of ensuring that your code is correct, ok.

So, this problem of managing concurrency is a general problem, there are several solutions to this problem, alright. And, you know the problem is in general a complex one, because you know the different kinds of situations and different kinds of situations require different types of solutions and responses from the programmer. But, one common response that most programmers use is what is called locks, alright.

So, what are locks well locks allow you to implement this mutual exclusion, right. So, we want to say that two calls to the insert function should be mutually exclusive, right. So, for example, in this ok; so, let us say I wanted to say yes, alright So, let us say the calls to insert should be mutually exclusive, but actually is it to just say that the calls to this function insert should be mutually exclusive; do I really need mutual exclusion on the whole function?

Well, I just need mutual exclusion on these two statements right A and B. If I could ensure that these two statements execute in a mutually exclusive way I would be done. It does not matter if these are you know mutually exclusive or notm ok. If these two statements had occurred in a mutually exclusive way I would have been done.

So, I need some instruction that allows me to say that these two statements are mutually exclusive other things I do not worry about. If you had made the entire thing mutually exclusive would that are worked?

Student: Yes sir.

Yes, yeah it will work because it still mutually exclusive except that you are you are constraining the system more than it needs to be, right; so, which can cause performance loss, right. So, these two statements could have executed parallelly on two CPUs, but now you are made the mutually exclusive. So, you know they cannot execute parallel, so you have civilized more things, right.

So, in other words to do you know called as a serialization. So, you know when you make something mutually exclusive you basically serializing the calls or to this or execution of the functions. The one thread is executing this, another thread is also executing this instead of allowing this execution we are serialized that, right. So, that is mutual exclusion, right.

So, you could have put the lock entirely, but the serialized more than required. So, you want to put a lock around these two statements that is one thing. Also do I need to serialize all calls to insert or can I do something smarter, can I say now let us have multiple lists and so, when I am inserting to the same list only then I need to serialize, right. If I am inserting a one third is inserting to this list and another third is inserting to that list, I do not need to serialize, right; so, I need to be able to represent that, right.

So, it; so, serialization does not occur only at the code level, it also depends on the data. So, let us say the insert function was taking an argument as a list in which you want to insert then you want to say oh you know hey only if the L1 and L2s are equal do you need to serialize, if they are not equal then you do not need to serialize. So, you whatever you are you know whatever this abstraction is needs to be able to do this, alright ok.

Also, I have said that this is insert, but let us see there is another function called delete right. So, once again you know delete will have some similar code that will involve two or three pointer manipulations before it gets complete. And so, you may want to serialize not just inserts with respect to each other, but also insert and delete. So, it is not just insert versus insert it is also insert versus delete, right. So, all these things need to get serialize with each other, right.

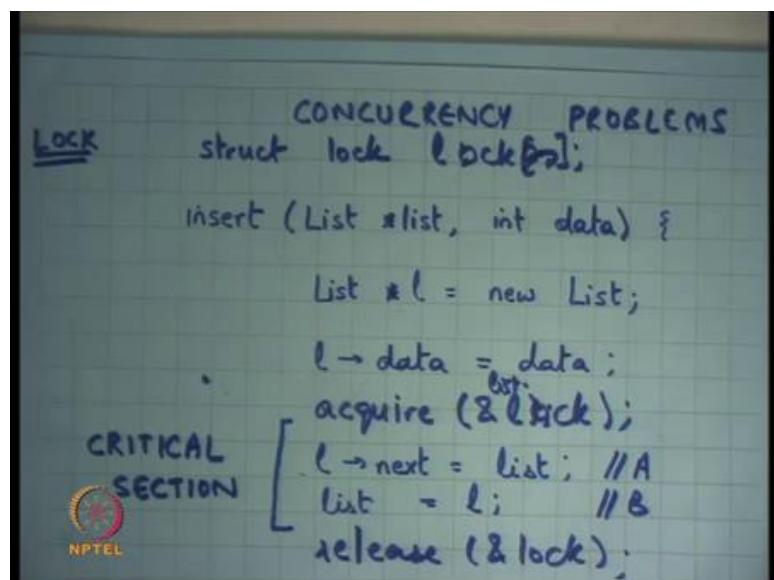
So, it is not just saying that you know this function should be serial with respect to itself, this function should be serial with respect to some other functions and I should be able to specify which other functions. And, it is not just functions; it is basically saying these statements should be serial with respect to those statements, right. So, I need some abstraction to do that right and locks one such abstraction and I want to talk about it in the next lecture.

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

Lecture - 21
Locking

Welcome to Operating Systems lecture 21 all right.

(Refer Slide Time 00:29)

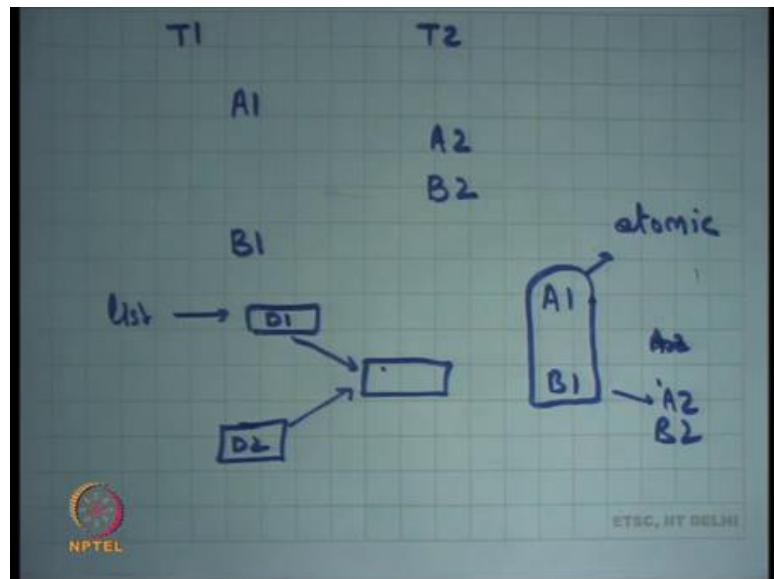


So, last time we were looking at concurrency problems right. And, we said let us we looked at this code, which inserts an element data into a list right and let us say this is a link list. So, the way this works is you declare a local variable, you allocate some node of type list from the heap, you said it is data, you said it is next pointer to the list, the list that exists that is even as the first argument and you update the list right.

So, you are basically updating the data in the front of the list. Now, and we said that look these two statements let us call this statement A and let us call this statement B; these two statements are needed to execute in an atomic fashion. So, they while the executing they should not get interrupted or nobody else should be touching this list while in the middle.

So, it should not happen that in the middle of this and other thread starts executing insert, if it does that then bad things can happen right. And, the bad thing in this case would be that the list structure is no longer preserved.

(Refer Slide Time 01:57)



So, for example, we saw last time that if it so, happens that thread the two threads T1 and T2, one thread executes A1, the other thread executes A2 and let us say it executes B2 after that and then this one executes B1 right, then what can happen, you had a list and thread one will insert it is data, let us say this is D1 dat A1.

And, thread two will insert it is data to the same list right. And, you will have something like this. So, list will be pointing here, and this is no longer a list now right. So, bad things like these can happen. Similarly, you know any other schedule which you know A1 A2, A1 and D1 getting interrupted and another insert getting called on that list bad things can happen.

So, another bad schedule is let us say A1 A2 B1 B2 all right or anything on this one. So, these are all bad schedules, and this basically corrupt your data structure right or violate your invariance right. So, what did you want, you really wanted that you know A1 and B1 should execute in a sort of non-preemptable way right? So, or in an atomic way ok.

So, if it is possible that while A1 and B1 are executing, now either you know no other call to insert should start or no other call to A and B should start. So, A2 should either be

here, or A2 and B2 should have been before A1 and B1 right that is what you want ok, all right.

So, in other words you want to say that these two statements A and B should execute atomically, and they should execute atomically with respect to something with respect to themselves right. So, they can be other code that is running simultaneously, but as long as that code is not touching this list it is right, but if that code is likely to touch this list, then it should not be now that code in this code should be atomic with respect to each other all right.

And, so we said that there is a there is an abstraction called a lock right, that we are going to discuss today. So, what are some so, when we are designing such an abstraction, what do we need to take care of firstly, in this function these are the only two statements that need to be atomic right, it does not matter what happens here right. So, it is only A and B that you need to protect.

So, you need some way of specifying that this is the area that is atomic, you do not need to make the entire function atomic number 1 right. Number 2, it will be nice if you could say that you know operations on this list need to be atomic with respect to each other, but operations on list 1 and operations on list 2 do not need to be atomic with respect to each other. So, you know insert on list 1 can happen concurrently with operations on insert on list 2 it does not matter right.

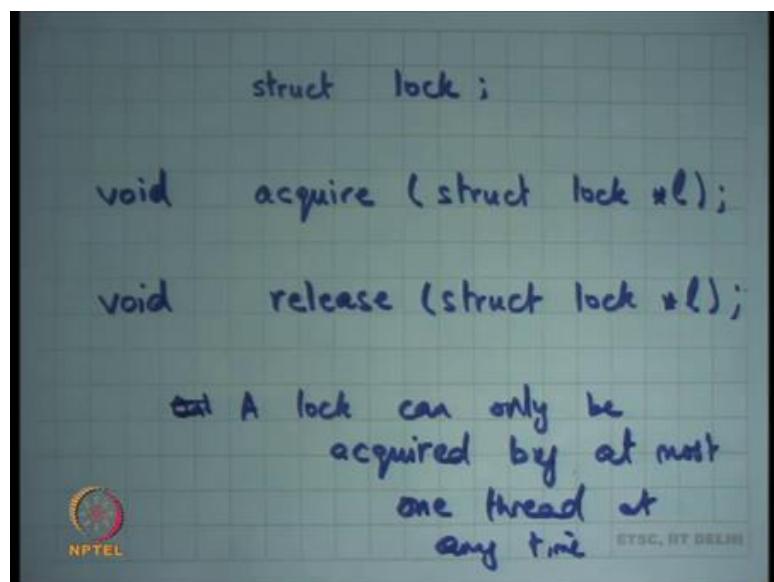
So, you would like to add that kind of flexibility, that allows you know better concurrency and better performance in general right. And, thirdly it may not be just this function insert right. So, insert it should be atomic with respect to each other or these two statements should be atomic with respect to themselves, but these two statements may need to be atomic with respect to some other statements right, but let us say there is another function called delete right.

So, delete and insert are happening concurrently, then say similar situations can happen where the list becomes corrupted right. So, it is not just these statements being atomic with respect to themselves these statements may need to be atomic with respect to some other coordinate system right. So, that is you know that is the so one has to designing abstraction that allows you all these different flexibilities right.

So, many different abstractions exist to solve this problem all right. And, you know as you go along in this course or in future courses you are going to come across, different kinds of distractions, different programming languages have different abstractions you know C will have a different kind of.

So, Java or something like that would have different kinds of support and different languages have different kind of support but locking or locks are one of the most common types of abstractions used to solve this problem all right.

(Refer Slide Time 06:33)



So, what is the lock? All right. So, a lock is defined by a type let us say the title struct lock right, or any other type there should be a type it says it is a lock right, and there should be 2 functions defined on this lock; one is called acquire right, which basically represents that the lock has been acquired all right. So, I am going to discuss what that means, and release all right.

So, this variable, this structure lock is a stateful structure and it represents two states; either it is locked or it is unlocked, or the other way to let us say it is either it is required or it is released right. So, that is the you know so, lock has 1-bit state of 1-bit state, which says whether it is locked or not.

The function acquire changes the state of that lock from unlocked to locked, but the precondition being that the state should have been unlocked earlier right. So, if it is

already locked then acquire will just wait till it becomes unlocked and as soon as it becomes unlocked it will turn into locked all right. So, that is the semantics acquire once again.

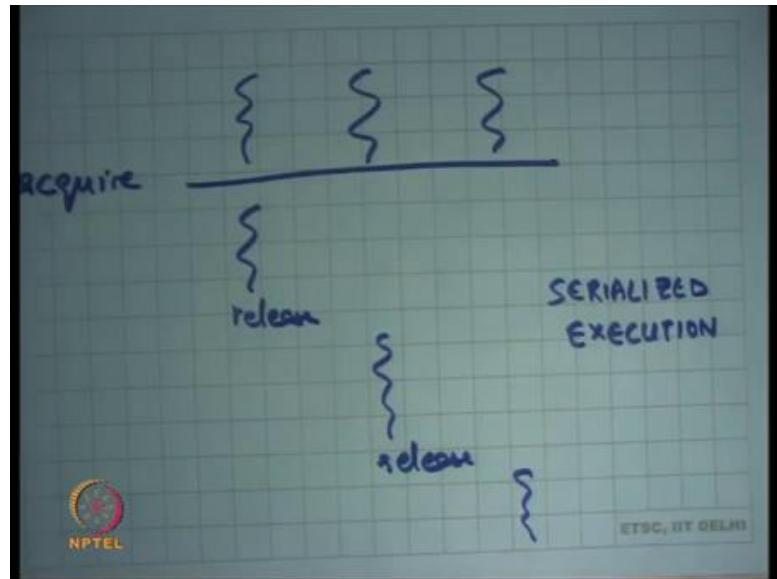
Acquire will set this state of this lock to acquired state or locked state, but it requires that earlier it should have been in the unlocked state or release state right. If, it is already locked then it will keep waiting till it becomes unlocked and after it has become unlocked it will set it to locked right. And, release will just make set it to unlock all right. So, that is basically it all right.

And, let us assume that these functions are you know are implemented in some way, how they are implemented we are going to discuss this later, but so, what happens is if two threads try to acquire the same lock simultaneously and let us say initially the lock was unlocked. So, let us say initially the lock is unlocked and two threads try to acquire this lock simultaneously only one of them will be able to lock it and the second one will keep waiting all right. So, that is abstraction.

The second one will keep waiting till what time till the first one calls release all right. So, in other words there is an invariant, the invariant is only a lock can only be acquired by at most one thread at any time. It is only one thread could have acquired the lock at any time; the second trade will have to wait.

Only out of the first let us say there are 5 threads or n threads then other n - 1 threads will have to wait, only when the first thread calls release or the is an somebody else going to get a chance to get the lock all right. So, what is it is doing? It is basically doing it is if multiple threads call acquires, then it is serializing these threads, it is saying multiple thread have try to call acquire.

(Refer Slide Time 10:14)



So, if multiple threads let us say call acquire and let us say they come at the same acquire point. From A1 let us say only 1 thread will get to run, only when it calls release then the other thread will get to run, when it calls release the third thread is going to run right. So, that is serialized the execution, but in a serialized execution only for the region which was within that acquire and release calls all right.

So, let us see what will happen in this case? So, we had this code, we had this function called insert that insert data into a list, and we want we were not happy with the fact that this code could be executed concurrently by multiple threads. So, what do you want? You want to serialize the execution of this code right.

So, one way to do that is define a lock all right. Define a lock and initially let us say the state of the lock is unlock and just put lock calls to acquire the lock and release the lock right. I think there is let us say let us call it lock. So, acquire lock and release lock all right.

So, this makes the code correct right, because if there is one thread, it will come here, the first thread will be able to acquire it, violated the inside this and other thread who tries to execute the same code will not be able to acquire the lock. So, it will have to wait. So, we have disabled concurrency in this region right.

So, this region which is protected by locks is also called the critical section right. So, critical section is a region of code that needs to be serialized and its execution and the locks are basically allowing it to get serialized yes question.

Student: Concurrent threads which you are talking they are on different CPUs or they can (Refer Time: 12:40) same CPUs.

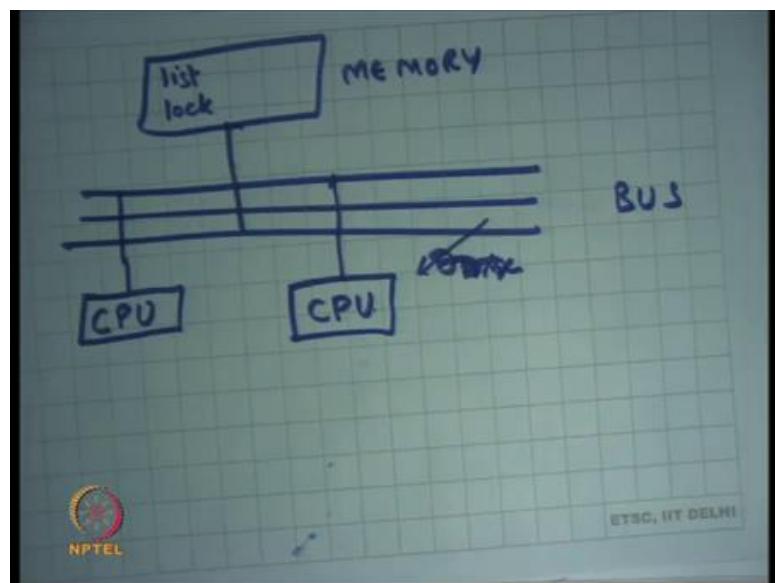
Right good question. So, what are these concurrent threads that is which I am talking about, are these on the same CPU or are these on different CPUs?

Student: (Refer Time: 12:49) same [vocalized- noise].

Same 6 ok, 10, 15 people different right, actually it does not matter all right. It can be on same CPU it can be a different CPU, let us look at how threads work right. So

Student: Sir, if is on the same CPU and if suppose I have acquired a lock and then it gets (Refer Time: 13:12).

(Refer Slide Time 13:20)



All right. So, let us look at what happens? Let us say so, let us say there is one CPU right, and let me just draw the diagram that we have, this is the bus and let us say this is the memory right. In the memory there lives a list right and what is happening was that this CPU was trying to update the list right.

And, while it was in the middle of updating the list it is possible that a timer interrupt comes right and the thread that was executing gets preempted or it gets switched out. And, another thread gets to run in which case what will happen is while I was executing at this line, an interrupt comes I get context switched out the other thread gets to run and have the same problem as earlier right. So, this problem exists even with one CPU.

If you have multiple CPUs the problem still exists it may be it becomes a little bit now the probability becomes higher, because it is possible that one thread is executing here and another thread is executing on the CPU, they are both trying now there is no interrupt involved right. So, there is no interrupt involved, but both these threads are trying to simultaneously access list and the same thing. So, same problem can happen.

So, the problem is really about interleaving. The interleaving can happen either, because of switching out on the same CPU or the interleaving can happen either because of physical concurrency of physical CPUs on physical memory right. In either case the nature of the problem is in the same all right. I have a question for you. Does the problem exist for user level threads or does it only exist for kernel level threads?

Student: Both (Refer Time: 15:07).

It exists for both right it does not matter whether it is user level thread or a kernel level thread it exists for both, because at the end of the day the abstraction is that the thread can be switched out anytime and another pair can get to run. At every instruction boundary a thread is potentially switchable all right. So, it does not matter ok.

Student: Sir.

All right question.

Student: Sir, suppose we for example, you have given in case of multiple CPUs, sir it does this lock will say like stop all the other threads running in all other CPU is adjust.

So, now, so let us say what happens with the lock right. So, we said there is a list living in memory. Now, in our new abstraction we also have this extra variable called lock living in memory all right. And, now one of these both these CPUs if both these CPUs try to acquire the lock in the same time, somehow this acquire function has been

implemented such that only one of them will succeed the other one will start have to wait right.

How it is implemented we are going to discuss in a minute? But, let us just say that there is this magic function called acquire that allows us to do it all right. So, assuming that there is such a function you know life becomes easier, I can put a lock around my critical section all right.

In fact, notice that this abstraction allows you to say that this is the critical section I do not need to say the entire function is a critical section. So, I can you know, I can just protect this area and these lines can execute concurrently. So, I do not need to you know impede performance on the other lines.

The other thing is if let us say there is another function called delete and delete is going to do some manipulations on the list also, then what you need to do wherever you doing those manipulations you need to use the same lock there right, you will make sure that you using the same lock in both in certainly. So, that way you can prevent you can make sections of code atomic with respect to each other by using the same lock around those section those critical sections right.

Thirdly, the third thing I said was I may I want to say that look this list and this list do not need to be mutual do not need to be aware of each other, they can proceed concurrently. So, in this case you know am I preventing that so, if I insert in one list and if I insert in another list will they get serialized.

Student: Yes (Refer Time: 17:36).

Yes, right they will get serialized, because A1 lock and you know I am calling insert on 1 1 and I am calling insert on 1 2 the same lock is going to get taken and so, only one of them is going to go forward. Even, though the list was different I did not need to serialize, but it will get serialized is it wrong to over serialize.

Student: Yes, Sir.

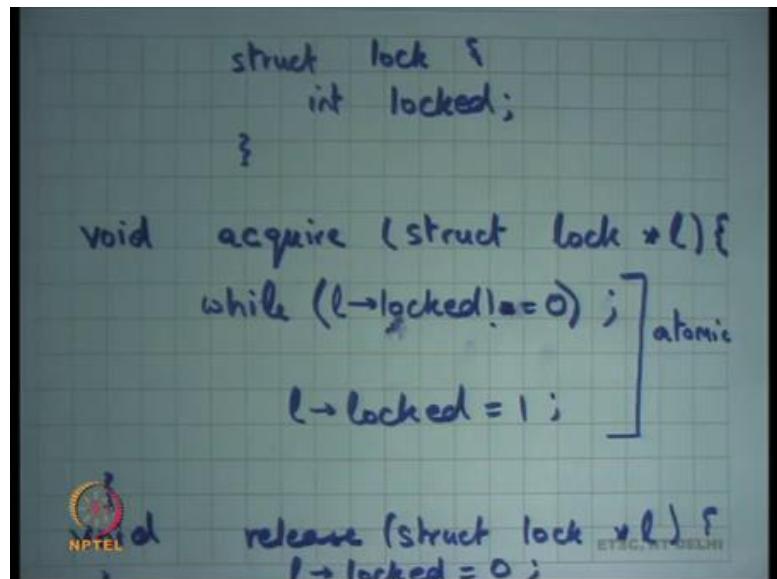
It is not wrong.

Student: The program.

The program remains correct, but performance gets affected right. So, what would having a better way better way would have been let us have a lock per list all right. Let us have you know let us have an array of locks right or better still in the list structure itself I have a lock right. So, instead of saying acquire lock I say acquire list dot lock or something right.

So, in the list structure itself I have a lock. So, this is a lock for this list. So, if you manipulating this list take this lock, we manipulating that list take that lock, but these 2 locks are separate lock. So, you can do this concurrently right. So, that is a thing about lock they give you a lot of flexibility all right ok.

(Refer Slide Time 18:48)



Now, let us see how locks are implemented? So, what do we need to implement we need to implement a structure called lock all right? And, let us say I implement it by having a field inside the structure, which says that I am locked or not I said this is state full abstraction, it basically has one bit of state which say that I am locked or not. So, let us say this is the state will lock.

Student: Can we have lock on subset of data structure may be like tree it has a tree and I already want to lock the left structure (Refer Time: 19:15) is it possible sir.

Can we have a lock per subset of a data structure? For example, if I have a tree then I want to only lock one sub tree and not the other side of the tree is it possible, yes, it is

possible ok. How, it is done I mean we are going to see as we go along the course all right.

So, let us see why acquire struct lock star 1 right. So, I need to implement this was one way to implemented well it is one very simple way just check right. So, just check is 1 dot locked =0, if it is 0 then very good just set 1 dot locked =1 and return right.

But if it is not 0 then keep waiting. So, let us say to wait I just put a while loop here right. So basically I check if 1 dot locked is equal 0, if it is 0 then actually let us say not equals 0 right. So, while 1 dot locked is not equal to 0 I keep spinning as soon as I find it to be equal 0 I said it to locked right, but if it is not equal 0 it means it is already taken I cannot take it I need to wait right. And, the way I am waiting is I am just checking it over and over again in a while loop all right.

And, how do I implement release, I say 1 dot locked =0 all right really a simple. I do not need to do any waiting I just need to revert the state back to 0 right. So, that is acquired that is released is this correct.

Student: Sir, but how do we define the preference between functions like, suppose there were delete and insert and suppose locking was occurring.

Ok.

Student: So, it right needs the gate start first.

All right. So, question is how do we define the preference all right? So, preference let us say there are 2 threads one of them is calling insert, the other thread is calling delete we should get first all right. Well, the order can be completely arbitrary right, the lock have actually does not say that you know does not give an order.

In general, you would just from a performance standpoint what is the best policy well one big one big policy is first come first served, whoever comes first given the lock all right. Who whoever come second you know has to wait irrespective of what function you are executing?

Student: Sir, then things will not be predictable or not.

Then things will not be predictable from what happened in science sense of way yes. So, there is still some non-determinism. Even, after you have locks it is possible that this happens right thread 1 executes before thread 2, what is possible that this happens thread 2 execute before thread 1. And, both of them will result in different final values of your memory all right.

So, this is called non-determinism there is still some non-determinism in your system depends on who gets there first, but the thing important thing is it is correct, it is not wrong. Earlier it was wrong, because I was allowing this and this is wrong, it was corrupting my data structure. This was right this was right, and I am allowing both of them and that is they tell some non-determinism in your system right.

In general, it is a very interesting thing that systems need to be non-deterministic for performance right. There is a very fundamental thing, if you make a system very deterministic, you lose performance right. So, let us take an example, let us say I wanted to make my system deterministic.

So, I say you know this lock will always be taken in strict round robin order, for thread 1 will be I know how many threads there are let say there are 2 threads. So, first I will give it to thread 1, then I will give it to thread 2 and so on right thread 1, thread 2, thread 1 thread 2.

So, if thread 2 reaches first, then I will just have to make it wait, even though thread and let us say thread 1 is line behind for whatever reason, there is a thread 1 is executing on a slower CPU, let say thread 1 didn't get enough time to execute while I was doing it. So, you know for all these or you know any other reason, thread 1 or thread 1 in general has more code to execute then thread 2 right all these are possible cases.

In either case the thread 1 is lagging behind thread 2 and thread 2 reaches a lock first if you want to have complete determinism, then you will have to make thread to wait which is just waste right, because there is nobody who is actually executing in that region, you could have executed it correctly you could have led thread to execute first, but you have to make it wait.

So, non-determinism allows performance and lock abstraction in general does not thought nondeterminism. It allows you to provide correctness, but there is still some non-determinism in your system ok.

Student: Is that possible 2 threads are bucking on a same list and 1 thread acquired a lock, but the other thread releases the lock without acquiring it (Refer Time: 24:29).

Is it possible that one thread acquires the lock while the other releases the lock? So, is it possible for a thread to release a lock without acquiring it?

Student: (Refer Time: 24:39).

It is possible, but you know we was assuming that because they are threads, they are trusting mutually trusting each other right. So, it is the programmers mistake that he has you know he is called list without acquiring the lock, it says problem he will pay for it by some error in his program and unless he is doing it very carefully all right.

So, notice that because you are operating the threads environment, we are basically saying that we are trusting other threads. So, threads must acquire and release for put acquire and release around all critical sections appropriately, they should not call acquire twice without releasing one thread should not call acquire twice without releasing, release should not be call without calling acquire you know that is. So, this is these are all invariant that the program I need to maintain. So, the headache is programmers.

Student: Sir, but if the struct lock maintains the like a list of threads, which have acquired it then may be like in the implementation themselves we can check this quality.

Right. So, firstly, a lock cannot be acquired on multiple threads simultaneously right, a lock has to can be acquired only by 1 thread at a time right. So, it is not a list of thread that has acquired the lock it is only 1 thread that could have acquired the lock at a time all right. And, the other thing is when you are using the lock it is better to not rely on the implementation of the lock.

So, that you can change the implementation underneath, just look at the abstraction; the abstraction is there is a structure called lock and there these two functions acquire and release. Do not make assumptions about how they acquire and how the release is implemented? In general, that is how; that is how it is done? So, that you know the

implementation of the abstraction can be changed later at well right. So, this is the standard practice that you will learn the abstraction ok.

Let us come back to this implementation is this implementation correct go ahead.

Student: Sir can I context switch while I am locked like can I.

Can I context switch while I am holding the lock.

Student: Yes.

Yes.

Student: Sir, then the other threads cannot do anything, because since I have locked it.

All right.

Student: It means it is basically wastage of some performance.

So very interesting question, if I have a lock I have taken a lock and in the middle of the critical section, a context switch happens, another thread gets to run, if that thread also tries to take a lock, take the same lock, then we will just have to wait all right. Till the next context switch and there are some wastages of performance absolutely true.

How these things are handled let us talk about it in sometimes. Actually, discussion about whether a context switch can happen, while you have taken a lock or not really depends on your implementation of how you have implemented the lock all right. So, but in any case, it is correct right. So, let us talk about correctness first and then we are going to talk about performance.

Student: Sir, if like thread one sees that locked value 0 and that context switch will take place and the other thread will also see that like the value of locked 0 then.

Good. So, you are talking about a problem in the 6 implementations yes. So, everybody understands that a problem in this implementation.

Student: Yes, Sir.

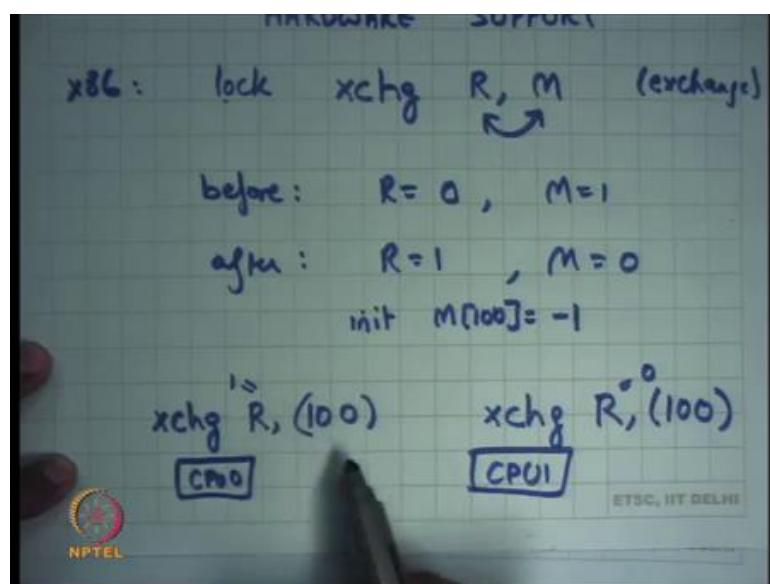
Right ok. The problem is one thread comes here, it sees that the lock =0 it is here a context switch happens or forget about a context switch let us say it is a multiprocessor system, and other thread also comes here, and both of them are here both of them take the lock you have broken your abstraction right the abstraction is no longer true. So, what did you want?

Student: (Refer Time: 28:05).

You wanted that this entire thing should have been atomic right. So, what am I done? I just you know I had some critical section that I wanted to make atomic, I said I am going to use locks to protect that critical section, but now I have changed the problem to say now I have to implement this lock, but now to implement the lock also I need some atomic away right.

So, I have not really made any progress except I made some progress in the sense that I have reduced the critical section from that large area, which can be arbitrary large to the small thing here all right. And, now the question comes, how does 1 implement this in an atomic fashion right all right.

(Refer Slide Time 28:55)



So, the answer is you need hardware support. So, there should be some instruction in the hardware that allows you to read and write a variable in an atomic fashion right. So, I am

going to take the example of x86, on the x86 there is this instruction called exchange all right. An exchange takes 2 operands: a register and a memory address.

So, for example, I could say and so, and the semantics are that it is going to exchange. So, this is exchange instruction solidification and it will exchange the contents of register in memory all right. So, if you know let us say initially so, before $R = 0$, $M = 1$, then after you will basically see $R = 1$ and $M = 0$ right. So, that is the; that is the semantic of the chain instruction.

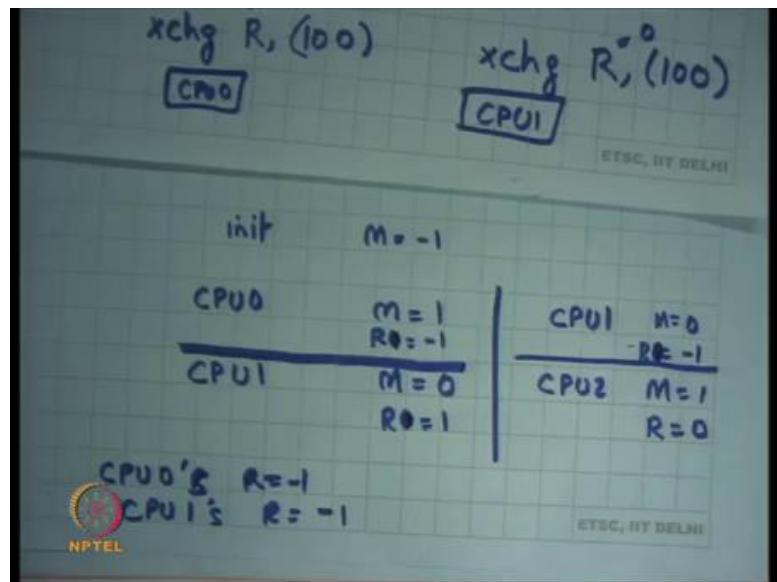
Notice that this instruction is doing two memory accesses in one instruction right. It is reading the old value of M and it is writing the new value of M . So, it is now it is both reading and writing in one instruction right. So, that is important. So, that in that sense this instruction is slightly special because it needs to read and write memory. So, it needs to make 2 memory accesses in some sense all right ok.

And, it is possible to make an instruction atomic in hardware. So, on x 86 there is this lot prefix. So, if you say lock exchange $R M$ then this instruction will atomically read the value of M and overwrite it with the new value, such that no other CPU can intervene in the middle right.

So, in what does it mean that if let us say there is CPU 0 and CPU 1, that vertex execute exchange on address let us say 100 all right and let us say I am saying. So, this is address and this is value let say R and let us say $R = 1$ and this one is saying exchange to the same address and let us say this was 0, then either this will happen before this or this will happen after this, it is not possible that the two reads and writes of one instruction get interleaved right.

So, for example, right so, let us say initially the 100 th location had value -1, then and let say CPU 0 executes before so, -1 comes to R and 1 goes to M right. So, this was initial of all right. So, let us look at this again in a bigger context so, all right.

(Refer Slide Time 32:11)



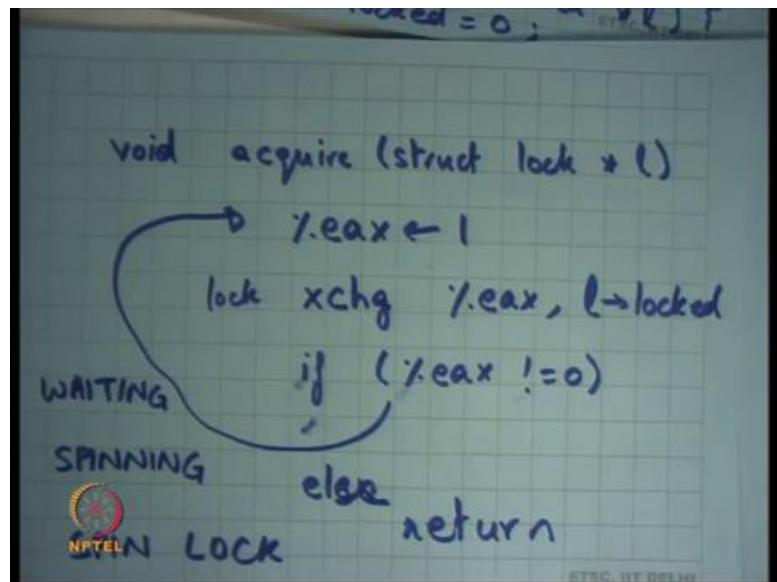
So, let us say initially $M = -1$ if CPU 0 executes then M will become 1 and if CPU 1 executes after that M will become 0 right and so, the final value will be 0. On the other hand, if CPU 1 executes first, then it will become 0, and CPU 2 executes later then comes 1 right. What is not possible is that they get interleaved right. So, what can happen if they get interleaved? Both of them read the old value of M and so, in this case you know CPU so, $R0$ would be equal to -1 and $R0$ will be equal to 1 in case of CPU 1 right.

In this case $R = 1$ will be equal to -1 $R0$ of CPU 1 = -1 that is now called $R0$. Let say $R = 1$, $R = -1$, $R = 0$ right. So, these are the only 2 possibilities, if 2 threads 2 CPU sideways execute exchange, they are atomic with respect to each other, what cannot happen is that both of them read the old value of M .

So, it is not possible that CPU 0's $R = -1$ and CPU 1's $R = -1$ not possible right this could have happened. If both of them read old value right and then they try to overwrite the new value and so, in that case exchange intervene atomic with respect to each other right this clear.

Basically, what I am saying is that the exchange operation itself is atomic with respect to itself and with respect to any other operation on that particular memory address all right.

(Refer Slide Time 34:25)



So, with that let us see how I am going to implement acquire, I am going to say void acquire And, I am going to you know use some register let us say I use register eax I am going to put a value called 1 in the eax. Now, I am going to do exchange eax with 1 dot locked right. Why was the memory address of 1 dot locked? And I am going to do in atomic fashion so, I use a lock prefix here.

So, what is and then I am going to check if eax is not equal 0 do what spin right else return right. So, I have taken the old value of lock into eax and put the new value of lock eax into lock. So, in 1 instruction in 1 go I have read the whole value and I put the new value into lot. If, the old value was 0 which means it was unlocked then I am done. If, the old value was 1 it means it was already locked. So, I did not change the value of lock either. So, I just keep retrying this is clear.

So, this my register I have set it to 1 I am trying to push this value 1 into locked, but I am; but I am also making sure that the old value of lock should have been 0 right. So, that is what I am checking here. If, it was if the value of lock was not 0, then it basically means that the lock was already locked. So, I keep trying again.

So, what is the difference between the old code and the new code? In the new code I am using this exchange instruction to read and write the value of locked in 1 go right. So, if the 2 threads which are trying to acquire this lock at the same time one of them is going

to hit the exchange instruction first. Over hits it first it is going to get the lock and it is going to return. The second one is going to see the value of locked to be one.

Student: The 1 which cannot acquire a lock well keep on performance exchange of operation again and again.

Yes.

Student: Then what will be waiting for then?

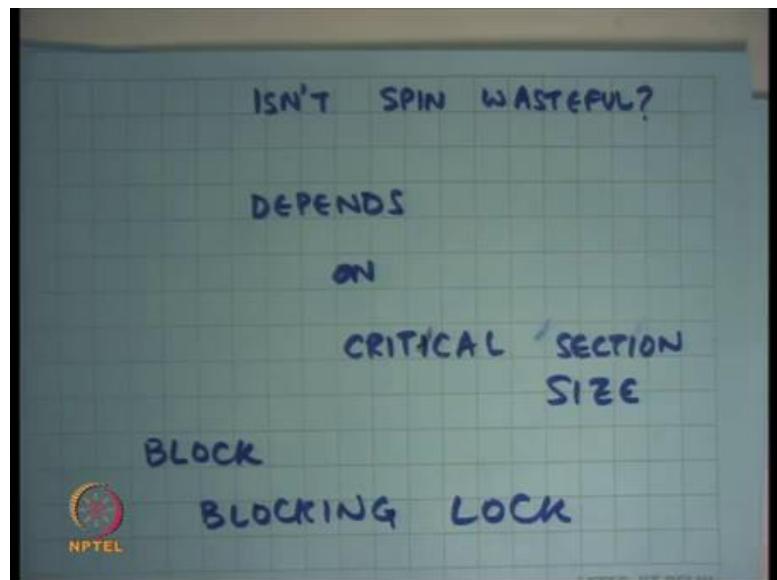
So good question the thread which is not able to acquire the lock we will just keep executing exchange operation again and again is not it wasteful dancer it depends and I am going to discuss, but let us first talk about correctness right.

So, good so we have checked if it was already locked. So, and this is correct right, because you have made sure that you are reading and writing this locked variable advance good. So, as I was pointed out one CPU will keep trying and other CPU will fall through. So, the one CPU that is trying is waiting and it is waiting in a loop all right. So, it is also called spinning all right. And, this type of a lock implementation is called a spin lock by the way how will you implement release.

Student: The previous (Refer Time: 37:52).

The previous one is right it does not matter just set it to 0 and that is fine ok. So, release a fine acquire only acquire need to change release was right. And, this is call, a spin lock ok. So, in this example what will happen if 2 threads come on acquire one of them is going to start spinning, the other thread is going to start running this code only when the first thread releases, the second thread which was spinning here is also going to get the lock and a second third is going to do it. So, basically serialized this operation of A and B and they basically doing it is in the exchange instruction right.

(Refer Slide Time 38:35)



So, now there is the question is not spin wasteful ok. So, what is the alternative? So, there is one answer which says can we disable interrupts would that be better just disable interrupts.

Student: And, then again enable it (Refer Time: 38:54).

And, then enable it at release. So, disable interrupts at acquire enable interrupts at release.

Student: Sir, not enabling and leave it once we have acquired the lock, we can enable the interrupts sir what is the problem?

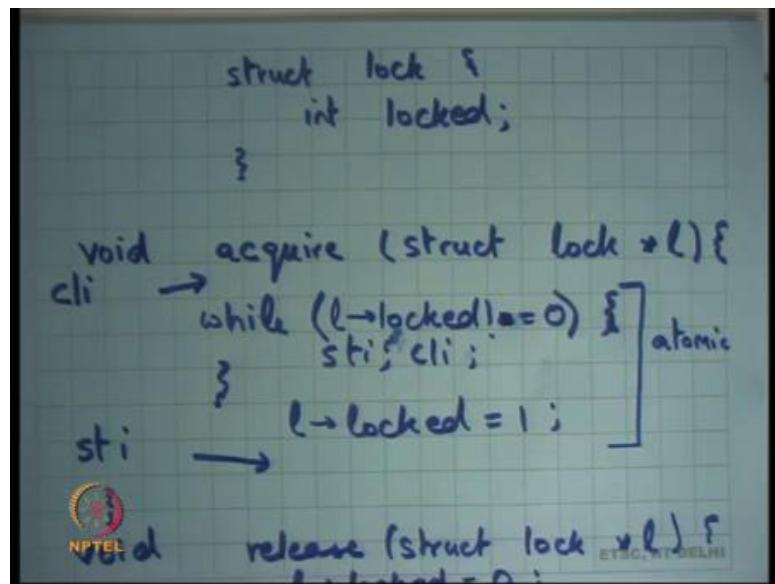
Ok.

Student: Is the critical state is there (Refer Time: 39:09).

Ok.

Student: Acquired.

(Refer Slide Time 39:17)



All right. So, here is here is one suggestion. The suggestion is in this code here right enable disable interrupts here. So, it called prior cli clear interrupts and then re enable interrupts here right. So, on x86 instruction is sti set interrupts right. So, call cli before and calls sti after and that should solve the problem.

Student: Sir, (Refer Time: 39:39) from the problem within the same CPU if we have multiple threads.

So, one answer is this will solve the problem only if there is a single processor. If, there are multiple processor it does not solve the problem perfectly fine all right ok, that is perfectly valid answer let us assume there was a single processor is this correct.

Student: Sir, it is some other correct, but (Refer Time: 40:01) exception then it will be (Refer Time: 40:02).

I have cleared the interrupts, I am spinning and let us say locked was equal to 0 and it was equal to 1. So, I am just waiting for it to get 0, if this is 1 part so, will it ever become 0, you know because I have cleared interrupts right timer interrupt cannot occur. And, so you know what is the hope? I am just basically deadlocked; it is not correct.

Secondly, you know a user program can definitely not do cli and sti right these are privileged operations. So, this privilege is only for the kernel, but even for the kernel this implementation is not correct.

What could have been correct? Well, if you really want to do it in this way one way to do it is inside this loop of while give the other thread of chance all right. So, just do sti and cli here all right. So, you are just disabling enabling disabling enabling interrupts. So, that there is another thread who gets a chance releases no deadlock right. This is actually a valid implementation of a lock on a uniprocessor; it is not a very very nice implementation right.

Student: (Refer Time: 41:12).

Student: Sir, this is also equally wasteful.

This is also equally wasteful all right ok. And, it does not work on a multiprocessor all right all right, but in either case in both cases we are doing spinning right, what is another, what is another way?

Student: Sir yield.

Yes good. So, the other way is yield. So, if one thread has figured out and it is not getting the lock you know as check the value of the locked variable it is 0, it is 1, it needs, it know that you have to wait why do you; why do you; why do you still going to going to CPU right. So, one way to do it is if we say that in that loop just call yield right. And, so, yield is going to suspend you and so, that way you may like other processes run, but does it really a better way.

Let us look at this example all right. In this case the critical section is very small it is just 2 statements, which maybe you know maximum of 10 instructions all right. Or let us say maximum of you know 100 to 500 nanoseconds to execute these two statements right. The other thread how long would it have to spin, it will probably have to spin for around that time which is you know 100 to 500 nanoseconds.

How long would it take to actually call yield, let us say 1000 or 2000 instructions, which is much larger right. It is much better to just wait for 10 instructions rather than call yield which is going to actually change my process structures and all that and it is going to cause much better. So, in this case it actually better to use spinlock.

In fact, yielding is wasteful right was yield there is a cost to yield. Imagine if there were 2 threads that try to acquire the lock 1 thread got it, the other thread call yield. Now,

these threads will very soon release it right. And, the second threads should have actually now got the lock, but now it is in this separate code path it is calling yield and then later it will get scheduled after you know the next time interrupt or whatever and so, it is wasteful spinning would have been better.

On the other hand, if the sizes of the critical section were large. Now, let us say you were executing 10000 instructions or if you were making some external access like a disk access right or some other slower device. If, you execute if your critical section is large in time, then it would be better to call yield, because for that amount of time you are just unnecessarily holding up the CPU all right. So, is not spin wasteful the answer depends and depends on what on.

Student: Size of critical.

Critical section size right. And, the basically the logic you want to use is if the critical section is likely to be large then use.

Student: Yield.

Yield right and the lock which uses yield is basically called a blocking lock. In fact, instead of using yield you could actually also use block right. So, what is block? Block basically just says that I want to block myself, because I know that this lock is not available. So, yield just says yield just puts you in the runnable state, which means on the next scheduling quantum you will again get to run.

Block basically tells operating system that I am not even in the runnable state right, I am in the blocked state, which means do not even schedule me and, at the release time the thread can say look at all the blocked threads and make them runnable right.

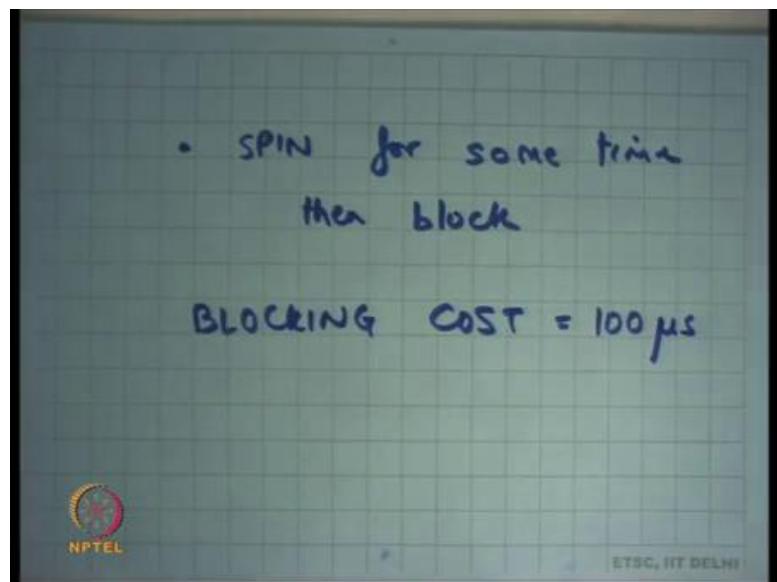
So, that is another way one is yield the other is block. Block is even more conservative; in the sense that if one thread is not able to get the lock it blocks, the thread which has the lock when it calls release is going to unblock all the block threads right. And, again there is some costs to block and there is some cost to unblock right and that kind of a lock is called a blocking lock right.

So, we saw a spinlock there is another lock called blocking lock, blocking lock will block the thread which call acquired and the release function becomes a little more complicated, because it unlocks the threads that have blocked on that lock right.

So, which one should you use? So, typically you will have both implementations available a spinlock and a blocking lock and depending. So, as a programmer you have to make a choice and you will basically look at your critical section size, typical critical section sizes and make a choice. It is possible that your code is such that some critical sections are very small, some critical sections are really large, and the same lock is protecting those critical sections possible right.

Some critical section is small and other critical section is large, the same lock is to protect needs to protect both those critical sections, what you do in that case? Well you can just say I will use a hybrid approach, I will spin for some time and then if I do not get a lock in that time then I will block right even that is a valid strategy.

(Refer Slide Time 46:38)



And, that is a strategy that is used in many lock implementations, which is spin for some time right and then block, while you were spinning you got successful you are done, but if you spin, if you spin for some time and you have not got the lock, it probably indicates that you know assuming that so, it is probably indicate that it is going to take a long time. So, it is better to now block right.

So, this is a sort of a hybrid approach adaptive approach where basically you at runtime figure out whether you need to spin or lock. So, how do you decide this sometime, whether it should be 10 nanoseconds, 10 milliseconds, 10 microseconds, 10 seconds? Once again it is a decision that you have to make depending on what is the cost it of blocking right? How long does it take to block?

Let us say blocking cost is let say there is this variable there is this; there is this quantity called the blocking cost, which is how long will it take to actually block this thread. So, let us say the blocking cost is let us say 100 microseconds right.

So, if you know that the blocking cost is 100 microseconds roughly speaking, then 1 1 approaches that spin for 100 microseconds and if you do not get the lock in those 100 microseconds and block. So, now you want to so, the maximum amount of time you are going to waste is 100 microseconds of spinning plus 100 microseconds of blocking total of 200 microseconds right.

Student: Like you said yield is costly right.

Yield or block is costly yes.

Student: Sir, but suppose even if I am spinning we cannot proceed until somebody has actually released it, but for that thread to run actually it will have some time to it will take the time to switch off the angle.

Yes.

Student: (Refer Time: 48:51).

So, can you repeat I did not understand your question?

Student: Sir, I am that you said we do not in for small ones we do not need to call v.

Yes.

Student: But I am saying that if suppose another thread was running suppose one thread was got (Refer Time: 49:12) in the spinning state. So, we need another thread to actually release the lock right.

No. So, I am talking about multiprocessor system. On a multiprocessor system you have to make this choice between spinning and blocking on a uniprocessor system.

Student: We are always (Refer Time: 49:29).

What is the better choice?

Student: (Refer Time: 49:30) blocking (Refer Time: 49:32).

So, how many says spinning? 3 blocking all right most of the others yes. So, blocking is the better choice all right on a uniprocessor, because if you know if you are in a uniprocessor, you know you are trying to get a lock spinning is not going to solve the problem right. You on the you know uniprocessor, if you keep spinning on uniprocessor you are never going to see that lock released, because nobody else is getting the chance to run.

It is only when you get switched out will somebody gets a chance to run and he will release a lock. So, it is anyways the switching is going to happen. So, might it will just block right. So, when I am saying there is trade-off between spinning and blocking, I am really talking about a multiprocessor system, on a uniprocessor it is very clear always block.

Student: Sir, but user level even if it is spinning it gets preempt effect.

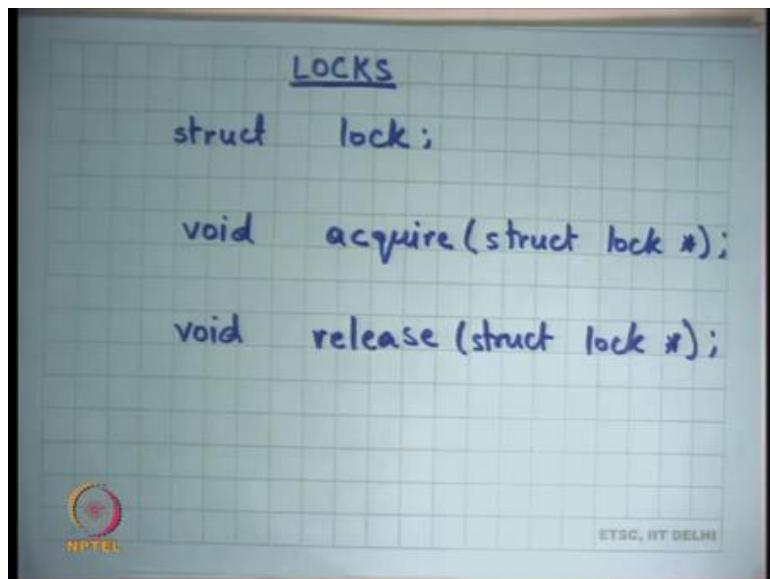
So, even in on a uniprocessor it is also to spin because you will get preempted. So, it is to spin, but from a performance standpoint it is better to block right. From correctness stand standpoint it is also to spin or even in a uniprocessor all right, ok, good all right. Let us stop here.

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

Lecture – 22
Fine-grained locking and its challenges

Welcome to Operating Systems lecture - 22. So, last time we were discussing locks.

(Refer Slide Time: 00:29)



And just to revise, locks is an abstraction, the abstraction has a type called struct lock so you can declare any variable of this type struct lock. And this type has two functions, acquire and release right. And the semantics of acquire and release are that only, the lock can only be acquired once at any time.

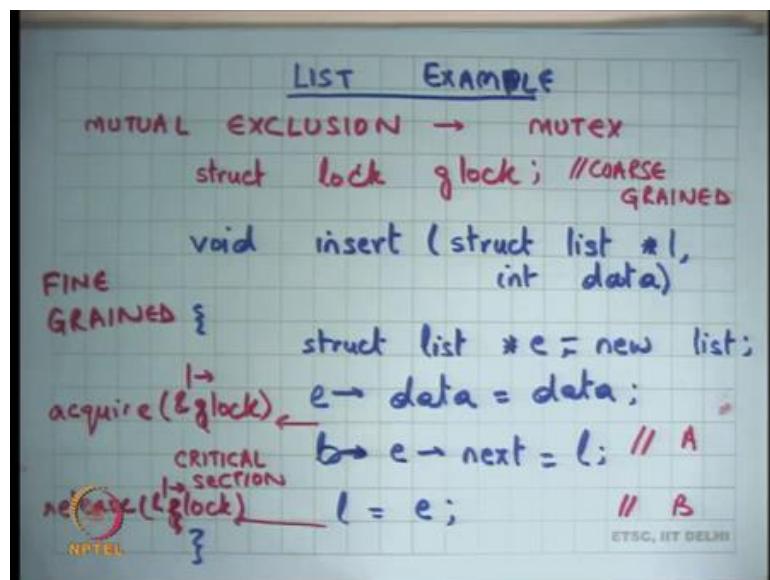
If another, if somebody if two calls to acquire a function are made before any release then you know, if two simultaneous calls to acquire are made or two calls are made without a release then one of them will not be able to acquire and only the other one will be. So, one if two threads try to acquire the same lock one thread will be able to acquire it and the other will have to wait, and the other will have to wait till the first thread calls release right. So, that is the semantics of a lock.

And we also saw last time how a lock can potentially be implemented. Basically, we use some sort of hardware support for an atomic instruction in the case of x 86, where we are

using the exchange instruction to implement the lock right. We also saw there are two ways to implement the lock, you could either spin wait, or you could either block right. Spinning is useful if you have a multiprocessor and you know they have critical sections are small.

So, you know spinning maybe the better option to do, but if you are on a uniprocessor or if your critical sections are large then blocking may be a more efficient thing to do, right. Because after all blocking is not free, blocking involves executing many instructions also and a spinning is going to finish very soon then maybe spinning is better right.

(Refer Slide Time: 02:05)



And then we were looking at this example of a list and we said look let us say there is a list there is a shared list *l* and there is this function called *insert*, that is going to insert this data element into this *l*. And here is this code which is just going to you know allocate a new list element and initialize its data and its next pointer and update the list right.

And if multiple threads are doing the simultaneously there is a problem because this area, let us say this is statement A and the statement B these are these need to be atomic executed atomically right. So, if two threads are executing these and the inter and the execution interleave on these two instructions then very bad things can happen and we saw some bad things that can happen.

And so of course, locks are meant to solve these kinds of problems and what you are going to do is you are going to put an acquire call here, and the release call here right. And because a lock cannot be acquired twice simultaneously only one thread can be active in this particular region right.

So, this region is also called a critical section right, and the and the lock and this act of actually using locks to ensure that the thread only one thread is active in the critical section is also called mutual exclusion ok. So, locks are a way to ensure mutual exclusion such and in fact, the locks themselves another name for the locks because of this word is basically mutex write that. So, a lock is also called a mutex in many scenarios alright. The other thing is you know you can actually because you know you can actually choose your lock variables.

So firstly, if you want to have mutual exclusion between multiple threads, does the lock needs to be visible to both the threads yes right. So, can the lock be a local variable here of course, not right because local variable is only said private. So, it has to be a global variable clearly.

But even if it is a global variable you know you have a choice you can either have a single, you know you can have a global lock, let us say struct lock g lock global lock and you can use acquire g lock and release g lock ok. So, you can do that, this is this perfectly correct code except that you know you are using one lock for all these insert operations.

So, if there are multiple lists, and you are doing in multiple threads are doing inserts on different lists simultaneously, they will become these operations will become mutually exclusive. So, even though there are multiple lists they did not need to be mutually exclusive, because you are using a global lock you have made them mutually exclusive. And so, this is called coarse grained locking right.

So, you are taking locks at a very coarse granularity, you are basically saying these are all the lists and I have one lock for all the lists right. So, irrespective of which list I am on you are going to take the same lock and you are serializing accesses to all the lists which is an over you know overkill you do not really need all that. So, what is the better option?

Student: (Refer Time: 05:25).

Have a lock per list right. So, we discussed this time last time. So, we can have a lock per list instead of a g lock let us say I have a lock variable inside the structure itself, and I can just basically take that and this is fine grained locking right. In general, coarse grained locking is easier to get correct, but its less performant, finer grain locking is a little harder to reason about, but it has potentially more performance, more concurrency.

So, let us take some more examples to understand this tradeoff between coarse grained locking and fine-grained locking alright. So, let us take this code and let us say you know let us say I am a bank and I am holding these accounts.

(Refer Slide Time: 06:05)

```
# define N 100
struct account {
    int account_id;
    int money;
};
struct account accounts[N];
void transfer (a , b) {
    if (a->money > 0) {
        a->money--;
        b->money++;
}
```

So, you know let us say I define this type called struct account and let us say each account has an account id or account number right and it how much money there is in the account right ok. And let us say you know I define some number let us say its 100 and I have these accounts and let us say this account is also as is, so these accounts are declared as this global array right.

So, these are accounts that I am holding, and each account is having some money and I am a bank and I want to provide this service to my customers called transfer right. So, I should be able to say void transfer, you know let us say account a to account b, I am

writing some syntax here. And let us say what it is going to do is if a.money > 0, then .money--, b.money++ and that is alright.

So, and let us say let me just write this in a different slide. So, you know let us say I want to provide this transfer functionality, that is going to transfer money from one account to another ok. Let us let me rewrite this on new slide because its little running out of space here.

(Refer Slide Time: 07:53)

The image shows handwritten C code on a grid background. The code defines a struct lock_glock and a function bool transfer(src, dst). Inside the function, it acquires the lock, checks if src->money > 0, and if so, decrements src->money and increments dst->money. It then releases the lock and returns true. If the check fails, it releases the lock and returns false. The code is annotated with red arrows pointing to the release and return statements in both branches of the if block.

```
struct lock_glock;
bool transfer (src, dst)
{
    acquire (&glock);
    if (src->money > 0) {
        src->money--;
        dst->money++;
        release (&glock);
        return true;
    }
    release (&glock);
    return false;
}
```

And let us also say that this transfer can either succeed or fail. So, let us say this is a bool transfer you know source to destination let me also rename the variable. So, that its clearer, and say if source dot money is greater than 0 source.money-- , destination.money++ say let us say I return true.

So, the intent of the transfer function is to transfer let us say one unit of money from the source account to the destination account and it can only transfer if the money was actually non negative right are actually positive and otherwise it is just returns false let us say right. So, let us say this is my transfer function. Now clearly, I mean so assuming that this function is correct if you are running it in a single thread, but if multiple threads are running this code then bad things can happen. What are some bad things that can happen ok?

So, one bad thing that can happen is let us say initially an account, the source account had 1 rupee or 1 unit of money in it. So, two threads come in, they both check that the money is greater than 0, because its 1 and they both try to decrement it and at the end of the day what you will have is that one account the account will have actually money equal to minus 1 right.

And so, two people would have been paid and the you know, and it does not make sense because money should not be less than 0 right. So, you are you are dealing with a nonnegative number here. So, there is a race condition here. So, what is happening is the between the check and update there is a race condition right, what you would have wanted is that the check and the update are atomic which means you would want to use some kind of a lock here right.

The other thing that can of course, happen is you know these statements themselves are not atomic right, minus minus and plus plus involves at least two or three instructions because this going to read it in to register two or three operations, read it in to register incremented and then write it. And we have seen this before that if these instructions that interlude bad things can happen. We call hits variable in the web server that we discussed before right.

So, you know these here are these problems with this ok. So, what will you do? Right, you will use a lock of course, and let us say one way to use a lock is I define lock, struct lock g lock once again it is a global lock and I just say acquire g lock and release g lock right and do I need this, this or do I need to do something else.

Student: (Refer Time: 10:52).

One more. So, I need to put a release g lock here I need to make sure that along all the parts of my program the lock is actually released by before I exit the function right of course, right. So, this is fine this code is correct ok.

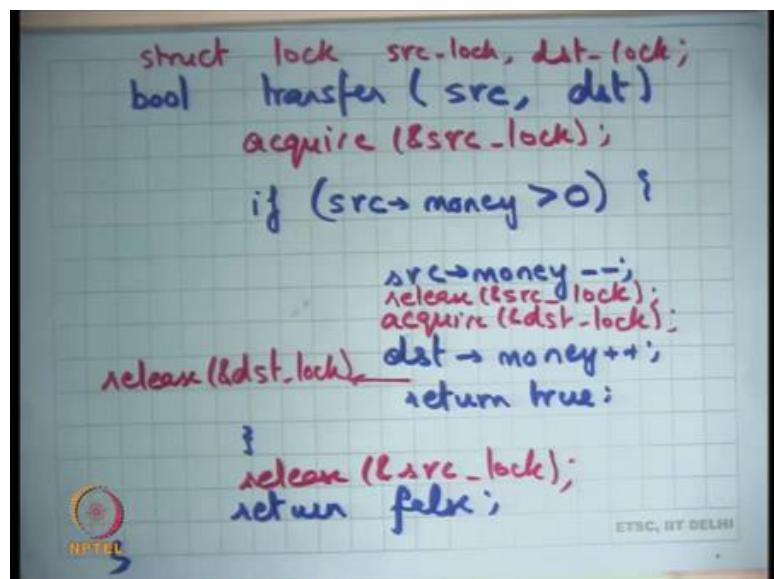
But is this the best way to do things well no because let us say you know I have a hundred accounts I want to transfer money to you and you know another person wants to transfer money to somebody else, you know we both of us will get serialized. Only one thread, one transfer instance we can work at any time it is not a very scalable solution, if your bank was you know this function is running inside an inside a server which is you

know connected to 180 atms it is not a very scalable solution right. So, what will you do? Right. So, you will.

Student: (Refer Time: 11:40).

So, I got two answers, one is put a lock at the source and the destination separately and the other is put a lock per account let us say let us try the first one the second the first one first right. So, clearly, we need to use fine grained locks, one lock is not good enough. We want to use more locks, so that there is more than concurrency right. So, let us say the first thing I am going to try to do is I am going to say lock the source separately lock the destination separately, alright.

(Refer Slide Time: 12:05)



So, let us say bool transfer, source destination if source dot money 0 ok. So, let us say I rewrite this code and let us say I have two locks now right. So, I have struct lock, source lock and destination lock let us say I tried to do this and let us say I say acquire source lock.

So, I am going to say here I am accessing the source. So, I am going to use a different lock to protect all operations on source and I am going to use a different lock to protect operations on destination right. So, what I am going to do is I am going to say acquire source lock and I am going to say release source lock right and let us say acquire right

yeah give me a minute. Destination lock let us say release destination, lock alright. And what do I need to do here?

Student: Release.

Release source lock right ok. So, let us say I do this. So, what have I done? I have made sure that all operations which are of this nature which is between source and source itself you are serial you are making them atomic and these operations are make are atomic alright ok.

So, what are some things that cannot happen? It is not possible that there is a race condition within the statement, it is also not possible that you check, and you know; so, these two statements cannot be concurrent with these two statements themselves, but is this correct?

Student: No.

No, why?

Student: Because it may happen that the source, the source of the parts would have been decremented, but the destination has not got the money now.

Ok.

Student: But now somebody may want that either off source or destination whoever has the money he can take; he will not be able to take.

So, here is one answer, he is saying that look you know it is possible at this point there is no lock held right. So, what can happen is at this point you have released the source locks, you know let us say I am transferring money to you my lock has been released I am about to take your lock, but let us say there is another thread which is just checking the total amount of money in the bank. At this point you will see you know the bank has one look one unit of money less right, that is not a and that may violate some invariants ok.

So, that is a valid point you know, because if there is a thread which is checking how much total money there is you know this transform operation is not atomic anymore. The whole transfer operation is not atomic anymore because you know there is a point where

somebody can observe the state of the bank and he will see an inconsistent state right ok. So, agreed, transfer is not atomic. Is there anything else that is wrong with this?

Student: (Refer Time: 15:48).

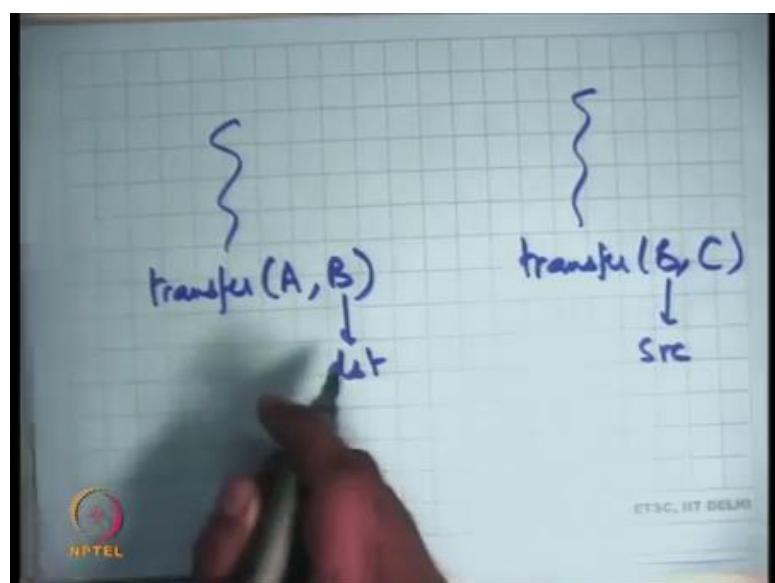
Right. So, let us just talk about that, let us just say there is a transfer in this function in this code in system. So, what if you know I want you know there is a thread. So, I wanted to one thread is trying to transfer money from me to you and there is another thread that is trying to transfer money from you to your friend right.

So, one thread has me as a source and another thread has and you as a destination and another thread has you as the source right. So, one thread has you as a destination and another thread has you as the source. So, what can happen? One of the threads could be operating on your account at this point and another thread could be operating at on your account at this point, is this greater is this I mean do you see a correctness problem?

Student: Yes.

Yes, because there is a race condition between one thread here and another thread here and the race condition exists because you are using different locks right. So, both threads are holding some lock, but they are not all in the same lock and so they can concurrently be accessing different, you know concurrently accessing the same data; however, right.

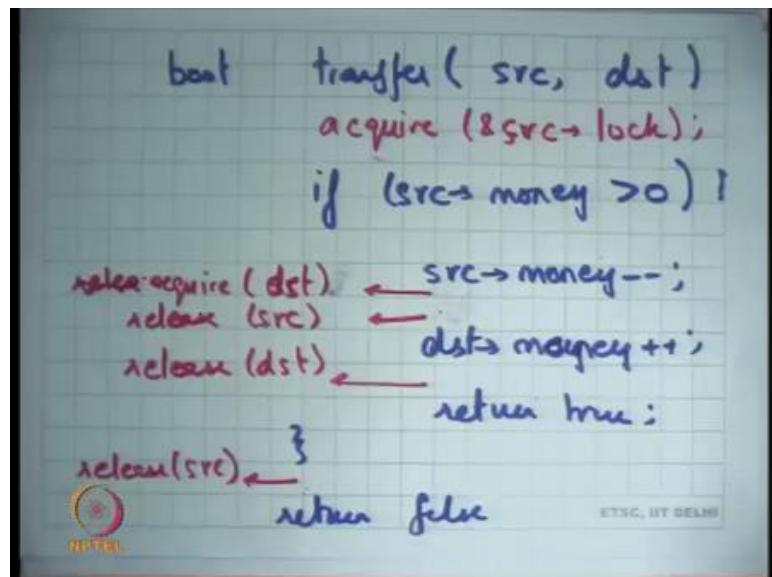
(Refer Slide Time: 16:57)



So, the problem is if one thread says transfer A to B, and another thread says transfer B to C; here B is the destination and here B is the source right. And this code does not ensure mutual exclusion between this code and this code right. And so, there is a problem, you wanted to also ensure mutual exclusion between this code and this code right, if it is possible for these accounts to be the same right.

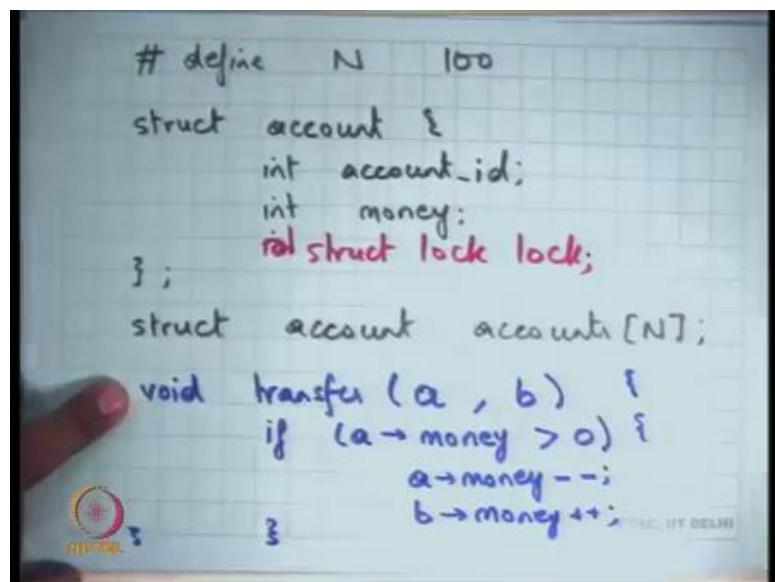
So, the problem really occurred because I was trying to have a lock per argument and the arguments could alias right. So, it is possible that one person's argument is another person source argument is another person's destination argument. So, the better thing is probably to have a lock per account right ok. So, let us see let us rewrite this. So, let me just rewrite this.

(Refer Slide Time: 17:57)



Bool transfer, source destination and now let me use right. So, now, what I am going to do here is I am going to say acquire, source struct lock ok. So, I now have an account a lock per account right.

(Refer Slide Time: 18:51)



So, what I what, the other thing I have done here is in this structure account I have another field called struct lock and so I am going to say acquire source struct lock. Where do I release it?

Student: Before the source release; after the source release.

So clearly you know I could alright. So, I could release it here, I could not release it here, I could release it here or I could release it here. I need to acquire another lock right I need to acquire the destination lock. Where should I acquire it?

Student: Before releasing; Sir acquire before both of them before the transaction starts and then release both of them before that.

So, one answer is acquire both of them here, and release both of them here and release both of them here.

Student: (Refer Time: 19:40).

Right. So, acquire everything in the beginning and release everything at the end just before exiting and you know so the somebody says that is a bad thing because no it seems its it may be correct, but it is not the best that you can do right.

Student: What we can do we can acquire source lock, then we can after source money minus minus we can acquire the destination lock and then after that we release the source lock and then after we have.

All right. So, there are many options, even I mean in the small code you can see there are you know many different sort of options and it is a little confusing, but one thing you can do is you can acquire the source lock here, you can release the source lock then you can acquire the destination lock, then release the destination lock and then release the source lock here right.

So, acquire, release, acquire, release. This has a problem that the transfer is no longer atomic right, because you have released the source lock then you acquired the destination lock, in the middle if somebody observes the state, he will see an inconsistent overall state. It is you know you may or may not care about atomicity of the whole transfer function right.

Sometimes you care about the atomicity of the whole operation sometimes you do not care about the atomicity of the transfer function. If the only service that the bank is providing is transfer, perhaps you do not care about the atomicity right. How does it matter?

If for some time somebody sees 1 rupee less in the account right, at the end of the day probably I am going to see I mean this everything correct right, but of course, you know typically you would want something like transfer to be atomic because you know there will be sum let us say you know sum thread which is let us say the sum thread which will just sum all the accounts and that may be running in parallel and so you may care about the atomicity.

(Refer Slide Time: 21:15)



So, the second option is that you let us say you care about atomicity of transfer. So, the second option was suggested was that you acquire source lock, but then before you release the source lock you acquired the destination lock, then you acquire the source lock then you release the source lock and then you release the destination lock. So, here is the; here is the suggestion you release no sorry.

Student: Acquire.

You acquire let me just shortcut it shorthand it, so you acquire destination you release source and then you release destination and what do you do here? Release source all right. So, notice that our code has become slightly, slightly tricky because at this point I have acquired one lock and I am trying to acquire the other lock right and I am I have to do that because this operation which involves accessing multiple sort of accounts needs to I have both, you know for atomicity I need to hold at least hold on to at least one lock I cannot just release both locks alright. So, let us see what can happen.

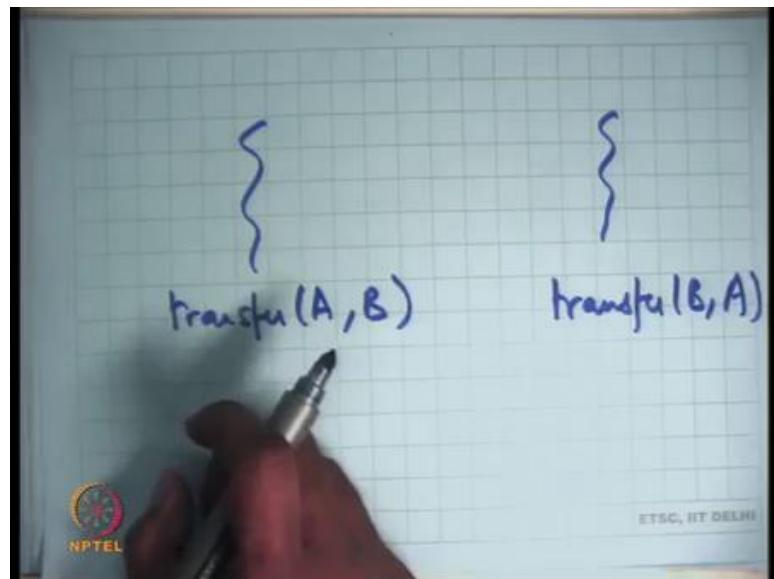
Student: Like one second.

So, if the source and destination are same then you can see a problem. You know you acquire the same lock and you try to acquire the same lock and what is happens? The thread deadlocks right it will never be able to acquire the same lock twice, that is

abstraction the lock. The same thread calls require twice it just dead locks it will never be able to proceed ok.

So, that you know you may say that is you know that you can check you can just say if source is not equal to dead then do this otherwise you do not do anything right. So, that is a very interesting observation, but let us just take this observation a little further let us say one thread is transferring account from me to you and another thread is transferring account from you to me right. So, what will happen? The first so let me just write it before I discuss.

(Refer Slide Time: 23:37)



Let us say this is doing transfer A to B, and this is doing transfer B to A alright. So, what will happen? The first thread will acquire as lock and it will try and may and it may be somewhere here and its possible at the second threads run simultaneously and it requires B's lock right.

Student: (Refer Time: 24:03).

Now, the first I will try to acquire B's lock and it will wait for the second thread to release B's lock. And the second thread will try to acquire A's lock and it will wait for the first thread to release A's lock. What do you have?

Student: Dead lock.

Dead lock because the first thread has acquired A's lock and is waiting for B's lock to get free. The second thread has acquired B's lock and is waiting for as lock to get free, both the threads are waiting for the other thread to release a lock, but they will never be able to none of the threads will be able to release any lock because they are both stuck at an acquire right and so, you have a deadlock right alright.

So, this is a big problem with fine grained locking, we you have you know big recall that coarse grained locking had had no such problem you just had one big lock you know, but now because you now you what you did was you somehow said you know maybe per account locks are better. So, you had per account locks, then you had some operations which involved multiple accounts.

So, when you have multiple accounts you need to take multiple locks at the same time when you have to take multiple locks at the same time then you have these problems of cyclic dependency right. There is a cyclic dependency because a there is a cycle because I am holding my lock and I am trying to waiting for another lock and that guy is holding the lock that I am waiting for and he is waiting for my the lock I am holding. So, there is a cyclic dependency in the lock acquisition order alright ah.

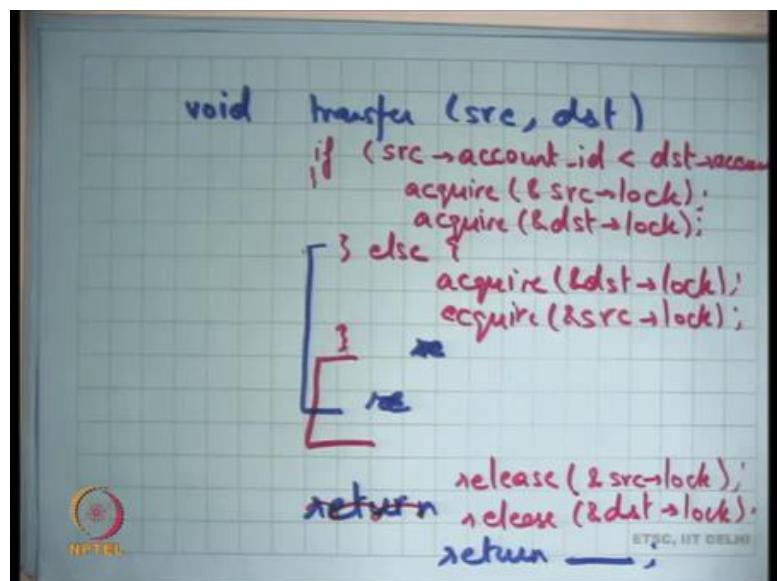
So, how do you solve something like this? So, clearly this code is not correct it is from a data race freedom standpoint. So, there are no races in this code transfer function appears atomic, but it has a problem that the function can actually deadlock, the system can actually deadlock. So, what is this; what is the solution?

The solution that is typically taken in an operating system and so these the deadlock is a very common problems seen in many different kind in scenarios, but in an operating system the typical solution to this kind of a problem is to have an order on the acquisition of locks.

So, you always say that if you ever have to do an operation which involves multiple entities and which involves taking multiple locks, then you will take those locks in a certain order right. So, one option is that you say that I am going to take. So, you and that order can be completely you know arbitrary, you can decide whatever order you want, but assuming that all if you are; if you are taking multiple locks and you are taking them in a certain order then they will never be a cyclic dependency right.

So, let us say one thread is transferring money from you to me to you and another thread is transferring money from you to me, then in both threads there will be an order right. Depending on what the order is either my lock will be taken before lock you, or your lock will be taken up before me. But it will not, never happen that one thread is taking my lock before you and another thread is taking your lock before me right. So, let us say I order it on account id right. So, if I order it an account id the correct version of this code may look like something like this.

(Refer Slide Time: 27:11)



Void transfer, source, destination and let us say you know I am let say this is my this is all my code which has all returned somewhere it has a return. So, at the end of it has return. So, I am not going to write the whole code, but let us just look at the; look at the locking behavior.

So, I may want to do something like this, if source dot account id is less than dst dot account id, then acquire source dot lock and then acquire destination dot lock; else what? Acquire destination dot lock and then acquire source dot lock and then after you have all these locks you can do whatever you like here, because you know these operations will be atomic with respect to each other and then at the end of the day you can say release, do the release need to have an order.

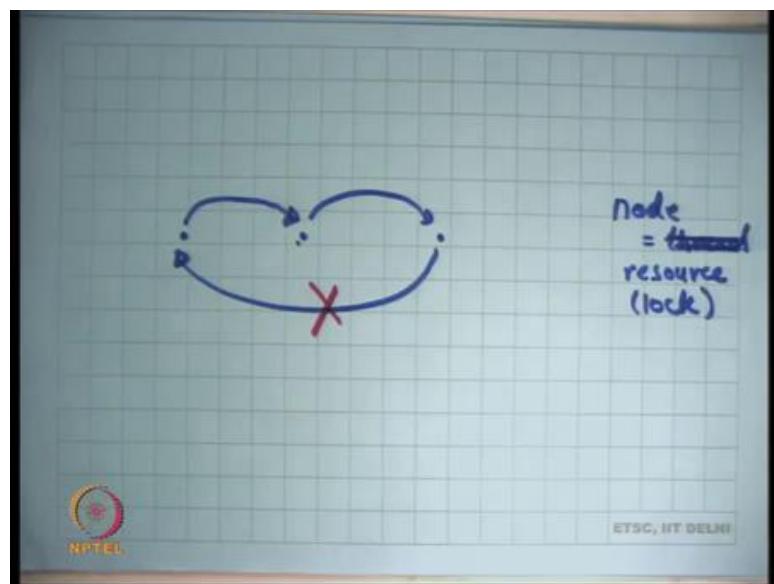
No, they do not have release does not need to have an order release is just a non blocking operation, you just say you know any order is fine and then you return. Now, whatever value you want to return here ok.

So, this code is correct, its fine grained each lock, each account has one lock and so and you have made sure that any operation that involves multiple accounts and needs to be atomic takes all the locks, all the corresponding locks and you have made sure that those locks are in a certain global order right. So, this order needs to be a global order and all threads respect that global order right.

So, let us say; let us say let us consider the same situation, let us say I wanted to transfer money to you and you know in one thread, and then there is another thread that wants to transfer money from you to me then in I in both cases let us say you know my account id was smaller than yours then both cases my lock will be taken before yours.

So, let us say the first thread takes my lock and then he is the second thread will also try to take my lock first and so it will block right there right. So, the second lock will not be taken, I mean so there will not be a cyclic dependency in other words you know basically a deadlock occurred.

(Refer Slide Time: 30:33)



So, let us say each of the, each let us say if I were to draw a graph and each thread. So, and each node in this graph is equal to a thread right, and let us say let me draw edges in

this graph, if I hold I am I need a resource if I can potentially request for a resource that is held by the other thread right. So, I draw an edge between two nodes if I can request for a resource that is held by the destination node.

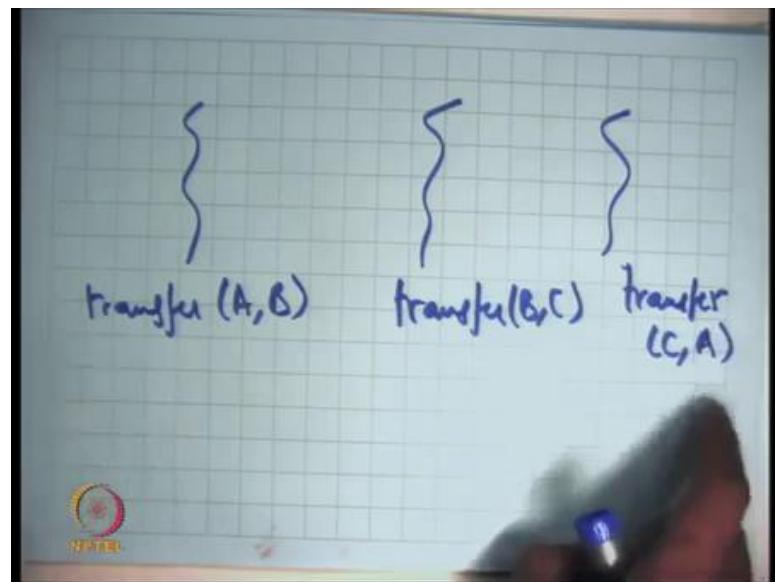
So, if the source thread can request for a source that is held by the destination thread, I draw an edge and the deadlock could only occur if let us say. So, let us say this could request for this and let us say this could request for this right. So, deadlock can only occur if there is cycle in this graph. What I have made sure is that these edges can only be forward edges because I have imposed a total order on the resources.

So, I can never be holding a resource. So, let us say these nodes are now resources right. So, the other way to look at it is a node is a resource, and resources you know I am using the word resource for a lock in this case. So, I draw an edge from one resource to another resource, if it is possible for me to hold a resource and then request for the other resource alright. So, it is possible for one thread to hold one resource and request one other resource as an edge this is the directed edge from that. So, and a deadlock can occur if there is a cycle in this graph right.

So, in this in the example in our accounts a deadlock could occur because it is possible that one thread is wait holding account x and requesting for account y and another thread is holding account y and requesting for account x, but if I put a total order on these resources, on these locks and I say that it is never and I make sure that such backward edges never exist right.

So, if I disallow backward edges, basically saying then there can be never be a cycle. So, that is basically why there can never be a deadlock if you completely order all these sources. So, you make sure that your resources are acquired in a certain global order, you will never have a backward edge in your resource acquisition graph right. And so there will never be a deadlock. So, you can you know you can extend this example to three threads.

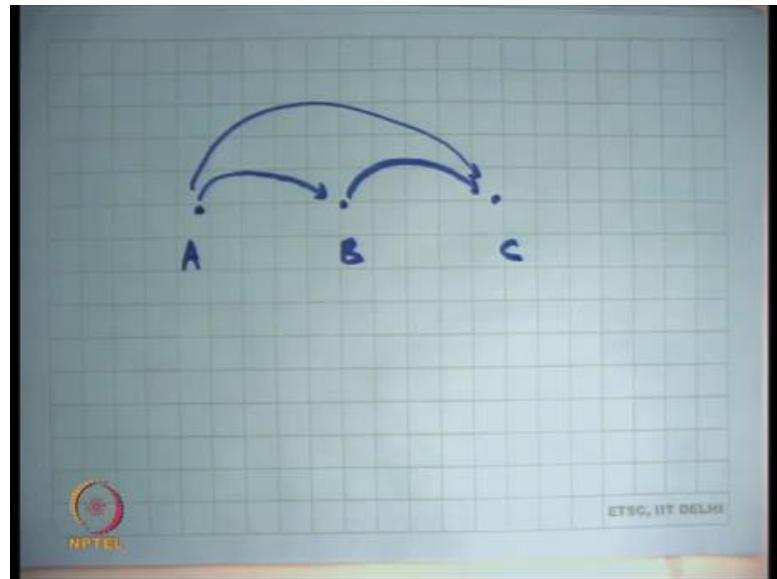
(Refer Slide Time: 33:11)



You know one saying transfer (A, B); transfer (B, C) and this one saying transfer (C, A) right. So, once again you know if you do not have a total order in your locks by account id then a deadlock can occur right, same thing I hold account A, wait for account B the other one holds for account B waits for account C and the third thread completes the cycle, he holds the account C waits for account A and all three threads; now dead lock right.

But if you have a total order then there will be no edge from C to A, there will only be edges from A to C right. So, if I were to draw the resource.

(Refer Slide Time: 33:55)



The let us say these are the locks and this is A's lock this is B's lock and this is C's lock this is possible and let us say A is less than B is less than C, this is possible this is possible, but this is not possible; instead I made sure I have written my code such that only this is possible right.

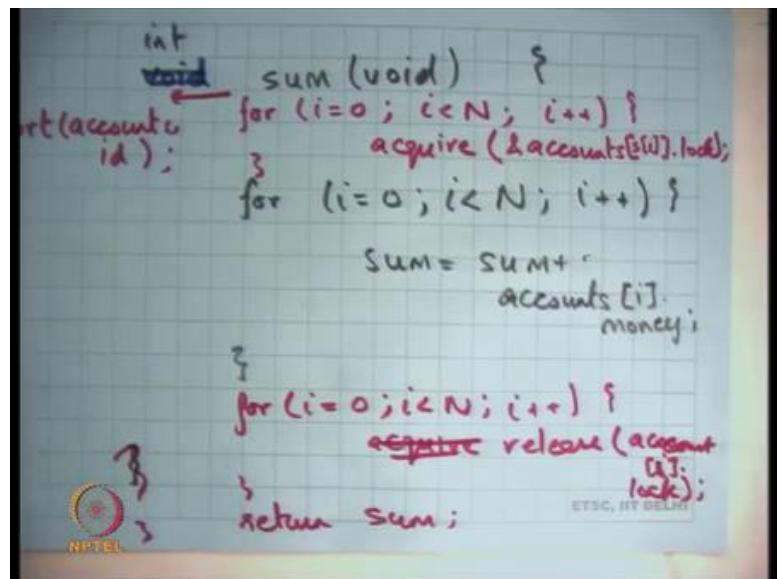
So, because I have ordered my locks, I can only have forward edges in my resource dependency graph and so they can be noted locks alright. So, let us see notice that the code that we had come up with was that will acquire source lock, then you know very carefully we will acquire destination release source lock after we will acquire destination lock and we will try to acquire the destination lock as late as possible and all that thing right.

But you know it looks very nice it would be nice if we could do this because it you know as you have you have probably thought that it is going to increase concurrency. But in practice it is not possible because you also need to order these locks in a certain way. So, you know basically we came back to the same solution that acquired both the locks in the beginning, right because you also need to check the order and then acquire the locks in that way right.

So, even though the most concurrent way to do things would have been acquired locks as you need them, but because you had to have a total order on the locks you could not do that, you just had to take all the locks in the beginning and in fact, you have to take those

locks in a certain order right. So, let us take another example, let us say in the same system there is another thread running that is called sum all right and let us say this right.

(Refer Slide Time: 35:27)



So, let us say the sum returns an integer and it just what it does is it just goes over all the accounts. So, for ($i = 0, i < N, i++$) $sum = sum + accounts[i].money$ right. So, that is it this is my code and of course, this function is also being done by a thread concurrently with other threads that may be running the same function sum or that may be running the transfer function alright.

And let us say you know I am going to make sure that you know I want to make sure that the sum stays constant. So, the total money in my account in my bank is should remain constant if we just we are transferring money across each other and so make to making. So, clearly if you want that to be true then the transfer function should have been atomic, and we have made that atomic itself. But can I just write the sum function like this? Would that be ok? No, why? What can happen?

Student: (Refer Time: 36:48).

The sum function could run concurrently with the transfer thread and the transfer thread could have decremented some money from some account and may not have incremented the money in some account and so the sum may be incorrect right. So, what do you need to do? You want to make the sum operation mutually exclusive with the transfer

operations and but so, in a coarse grained world it would have been very easy you would have had just one lock and both this transfer function and the sum function would have taken that lock and that would have made everything completely mutually exclusive. Now in the fine-grained world which locks did you take and when?

Student: All the locks, all the.

So, one answer is let us take locks for all accounts everybody agrees?

Student: We think is this two in order.

So yes, I think we need to take locks of all accounts because that if we miss even one account then you know our total sum will be wrong. Now, where should I take these locks?

Student: (Refer Time: 37:49).

Should I take the lock on demand around this?

Student: No, it has to be taken around (Refer Time: 37:55).

You, if you take the locks around this then you know you are violating the order right. So, here is an operation that just is not operating on two accounts in operating on n accounts right and it and this operation also needs to be atomics with respect to other threads. So, what are you going to do? You are going to take all the locks apriori then you are going to do the function and then go to release all the locks after that you cannot take the locks on demand because you need a certain order on the global order on the locks right.

So, what you?

Student: (Refer Time: 38:29) take the way that their accounts i dot id is always lesser than accounts i plus 1 dot id.

Sure. So, let us talk yeah. So, let us first look at what we can do. So, let us say so, the right thing to do would be for i is equal to 0, i is less than N, i plus plus acquire. Accounts i dot lock this is not correct either right because I am not obeying any order on the lock, I should have really ordered it by. So, you know what we will need to do is let

us say you have another array which sorts the accounts by id right and so and that sorted order is what we use here right.

So, you just take the locks in a certain sorted order and the sorted out for order is let us say in the order of the account id ordered by the account id and similarly you are going to just release it here. Alright here you do not care about the order right and return sum right. So, notice that the array itself is not necessarily sorted by the account id right, you just generate a permutation which is the sorted permutation s which is you know it sorts the accounts by this account id and then you basically do that right and so, you need to take them apriori.

Of course, now you can you know you may say, but you know what perhaps the better thing would have been that you know just arrange your accounts in a certain order apriori in the order of account id or allocate the accounts in order of the account id. And so that way you know you do not need to do the sort operation and you can probably even take the accounts in on demand in that case sure.

Student: So, that even in once we have sorted that now we can take the locks on demand (Refer Time: 40:52).

We have no t sorted the accounts we have just you just got a permutation. So, notice that I am just saying accounts si . So, s is a permutation, which is a sorted permutation now which basically says what will be the sorted order of these accounts.

Student: Sir (Refer Time: 41:27) we can also.

So, what you could do is you could say you know accounts si here and so then you can take the they take the.

Student: We have to make sure that we do not release we always hold on to one.

Right you just have to make sure that you always hold on to at least one lock right. So, that is true ok. So, the question is cannot we just say acquire account i and release account i right here ok. So, I see some head saying no. So, why?

Student: That is it we have to transfer from A to B (Refer Time: 41:44).

Right.

Student: And when A is less than I and B is greater than I then we have already added the amount of the previous amount of A into the sum, but now there is a transfer (Refer Time: 00:00) also then.

Right. So, basically you know let us say there is a transfer going on from account A to account B and now sum takes lock on account A and you know let us say before the transfer started and it got the lock and it will you know read some value then transfer happens then it tries to get an account lock on account B and then you know it gets it and so, it will get a wrong sum it will probably get it can even get you know something which is bigger than your actual amount right ok.

So, in general figuring out where to put the locks, how to put the locks, whether to put the locks or not is you know hard. In fact, lock is just one way of ensuring correctness right. Notice that locks are just basically what we did we said, this code is not correct if executed concurrently, let us make this code mutually exclusive. One way to make things mutually exclusive is locks and then let us use locks and then we said where to put the locks that itself seems to be a hard problem.

In fact, if there are other ways to make sure that the code is correct which does not involve taking locks where you can just structure your code in a very careful way such that without taking locks where you can actually make sure that code is correct alright and we are going to discuss some as the as we go along the course.

So, in general ensuring correctness and concurrency is hard, but and you know you would typically want to avoid things that are hard to understand because they are you know not just its hard to code, but it is hard to maintain overtime. Typically, programmers follow a discipline, and that is the locking discipline.

And the locking discipline basically says that firstly, you should think about your system as a whole and figure out where do you want to put the locks or how do you want to associate the locks. You want to associate a lock per account do you want to associate a lock per pair of accounts, do you want to associate a global lock with the whole all whole area of accounts these are all possibilities and you can you can choose one on them.

Once you have done that once you have decided that then you ensure that for all shared regions for all regions which can access this shared data or access the region that is

predicted by that lock that lock is always held before executing that region. So, if there is any code that can access a shared region then the lock corresponding to that shared region should be held, by that lock right. And if there is an operation that acquire that needs to touch multiple shared regions and needs to be atomic then locks of all these accounts should be acquired apriori or all these regions should be acquired apriori and released after that. So, this is so, and they should be acquired in a certain global order right.

So, these are this is basically the discipline that if you follow you will get correctness, may not be the best way to get correct correctness, but it is a reasonably good way of getting correctness. So, in the example that we saw there was transfer and there was some right in transfer also we did the same thing we said we are going to operate on two accounts.

So, let us take the accounts let us take that locks for both the accounts apriori then do the operation and then release both the locks. Similarly, in the sum operation we said you have to touch all these shared objects. So, let us take plot for all these shared objects apriori, touch all these shared objects and then release all these shared objects after that right. So, in both cases we were just using the locking discipline.

Student: Sir.

Yes.

Student: Is eventual consistence tolerable in OS?

Is eventual consistency tolerable in OS? So, what is eventual consistency you know these are. So, in general eventual consistency. So, eventual consistency has been is something that you know distributed systems people talk about, where you say that let us say the two people who are accessing the same thing eventually it will get consistent, but sometimes you may see inconsistent values right and you know so, let us so you know when you are designing a system you have to worry about what does consistency mean and what kind of consistency guarantees you need to provide to the user of your system right. Let us just talk in concrete terms before we know.

So, those are those become sort of much more sort of high-level discussions. Let us just talk in concrete terms it looks let us look at these examples and see you know what you need. So, in this case you know this is the consistency level you need, and this is how you do use it.

So, since we are talking about a bank you know if there is some thread is going to run, the sum thread is going to have to take all the locks and basically; that means, that all the transfer threads basically gets stopped for that amount of time right till the sum gets computed. Or, in fact, till the other locks get taken are there better ways to do that? Yes, there are better ways to do that. In fact, banks are not, you know bank accounts are not implemented in memory I am just using a toy example right you would typically implement bank accounts in a database and database has other ways to do concurrency control right because.

So, let us understand, I will just give you a brief understanding of why databases con the way the database does the concurrency control is different from how an operating system needs to do concurrency controls a database needs to acts you know operates on disk blocks because it also cares about persistence you know data should remain across reboots.

Disk accesses are much much slower than memory accesses right and so the overhead to do concurrency control is much smaller in a database. And so you can do more fancier things to do concurrency control in database and typical way of doing the concurrency control database of what is called transactions right.

So, we are going to discuss some of this later. So, databases do transaction con concurrency to a different way, an operating system cannot do transactions because you know it is too costly to do transactions for memory accesses right. And in fact, you know more recently there are there you know modern hardware is coming up with what is called transactional memory that allows you to do some of this.

And you know these are all advanced topics that we may have time to cover later in the course ok. But, by and large today locks is the way to do concurrency control and you are right I mean, if there is something like this, then the sum thread is actually you know basically bringing everything else to sort of halt for that amount of time. But that is basically what the price you will have to pay anyways.

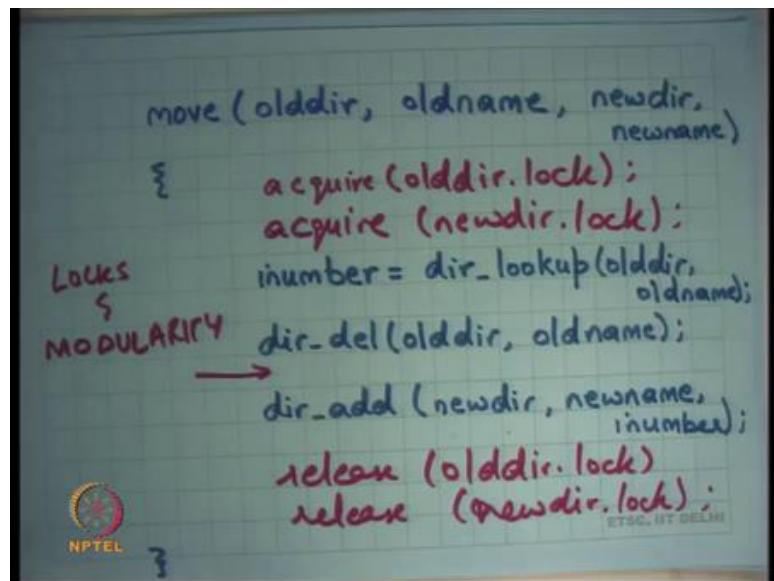
Good; no more questions alright. Let us stop and continue the discussion next time.

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

Lecture – 23
Locking variations

Welcome to Operating Systems lecture 23 right.

(Refer Slide Time: 00:30)



So, last time we were talking about locking, and we took this example this is hypothetical example of bank with many accounts, and with functionality like transfer, and sum and we said that look coarse grained locking can solve all the concurrency problems but coarse grained locking is not good, because it is serialize everything right. So, because it mutes in ensure that everything is mutually exclusive; it basically causes everything to get CU lies.

So, even if there are multiple processors only one transfer function will be able to execute anytime if you are using coarse grained walking. So, then we said ok, you know we should use fine grained locking and the question was how should you; how should you decide, how do use fine grained locking? So, this choice of how to choose where to use fine grained locking is a bit of an art. So, there is no, so I mean it is basically something that the programmer has to decide, based on what he feels is the right way of doing things right.

So, there is no one rule or to say that this is how you should fine grained lock in this program or that program, depending on the program you would not want to choose your fine grained locks differently. So, for example, you know yesterday we said that every account should have a lock so, there should be a per account lock. And any operation that require requires access to multiple accounts, you should take all the locks before doing that operations, all the locks for all the locks for all the accounts that are touched in that operation.

So, transfer operation touch 2 lock, 2 work rounds, you are going to take 2 locks, the sum operation touched all accounts you are going to take all locks right. And, we also said that one way to take the locks is to take it in an on-demand way. When, I say I take I take it in an on-demand way it does not mean that I released the previous locks right.

Because, this operation is basically an operation that needs to be atomic, we need to take all the locks at some point in time anyways, it is just that you can say that I know I will take the lock for the first count, then do some computation and then I will take the lock for the second count without releasing the count for lock for the first account and so on right.

So, you could do that, but we also saw that the locks have to be in a certain order to avoid deadlocks right. And so, the ordering the and the ordering has to be global. Once again you know the programmer has to figure out what the order has to be and the order will may be tied to your data structure, it may be tied to the semantics of your program. For example, the last time is we decided that we going to order it on the account ID of the account. And based on that we will take a priori all the locks needed for transfer we will take 2 locks, for some we will take all the locks, do our atomic operation then there is all right.

So, that was the that was a hypothetical example of course, let us look at another example, let us say I have a file system. So, as you know as an operating system one of the services that an operating system provides you is a file system. What is the file system? A file system is an on-disk data structure right.

So, a disk is nothing, but raw magnetic device which has lots of blocks and a file system is a data structure built on top of this sort of storage which allow and the semantics of the file system are usually the file system is hierarchical. So, you have a root directory and

then you have some names and each name may be a file or another directory and so on and so, you basically build a directory tree and that is basically what a file system is.

Now, you can imagine that there are multiple processes running in the system, multiple processes are making multiple system calls concurrently. So, one is calling read, another causing write, on different files, on same files all these are possibilities. So, question is the operating system needs to synchronize or make sure that operation accesses to the file system are correctly, you know correctly done and basically you know it basically means that operation should be atomic.

So, if there is an operation going on here and, in our operation, going on there, they should not appear interleaved at any point, because interleaving of those operations can cause bad things in your file system all right. So, you know one option is once again coarse-grained locking put a lock on the entire file system; you are safe, definitely safe right. But of course, that is not a very good solution you can imagine that your system will run at very very slow speed. Now, nobody will be able to access the file system concurrently only one person will be able to access the file system at the end.

So, what do you do? Once again choosing what locks to take is a bit of an art you may say let us have a lock for a directory, or you may say let us have a lock for file, or you may say let us have a lock for you know just very hypothetically. Let us have a lock per pair of files, you know if you figure out that most of the operations are actually occurring on pair of files.

So, why not you know have a lock in sensation per pair of files and if you are going to you know and do an operation between those 2 files or something but you know when in that case if you are going to touch one file then you would take all the locks in which for that file where that file belongs to a pair.

So, if you know for all the pair for that file you need to take a log so, that does not make a lot of sense. So, yes, I mean you know intuitively it seems like the best thing to do is basically take a file for lock a lock for file all right. So, some what I am going to show you is basically you know if you do this kind of fine-grained locking it hurts your program structure. So, if the program structure does not the modularity in your program actually reduces because of this.

Because of the locking behavior. So, let us say because of fine grained locking basically. So, let us say I have a function which looks like this, it says move so, just moving a file from one directory to another directory.

So, it says move this file name called old name from old directory and put it as new name in new directory right. So, that is the semantics of this function and what it does is it basically looks up looks up the disk block. So, let us say i number is the disk block or some identifier which is identifying the number at which this file is stored just looks up the old name in old directory, delete deletes old name from old directory and adds new name to new directory, that i number that you looked up right.

And, so this code is correct let us say when you run running serially. When there is only one thread that is accessing it, this code is also correct if you are having one big global lock that is protecting this entire function but let us say I have per file locks right. So, or per directly locks, so, let us say I have per directly locks. And, so what do I need to do I am accessing I am accessing the old directory, reading, and writing the old directory here. So, I need to have I need to lock this region with old directories lock, and I am adding something to new directory.

So, I need to lock this region with the new directory locked, but can I do these in isolation? Well no because you know I want perhaps I want my move operation to be atomic right, if I just say that over let you know that let directly delete do the locking inside it and I do not care about you know what locking it does inside. And, then let direct directly add do the locking inside it and I do not care then what happens is at this point here no locks are held and anybody is free to observe these directories or the state of these directories and at this point what you are going to find is that this file does not exist anywhere right.

And, so this the file system is in an inconsistent state at this point, you know so there is some there are some disk blocks that do they are not pointed to by anybody, neither by the old directory now nor by the new directory and that is an inconsistent state right.

In other words, you know if you do it in that way the move operation is not atomic right. So, what you what would you want you would again want to do basically something like this, you would say acquire old dir dot lock and acquire new dir dot lock right. And, then you will do this operation and then you will just release these locks.

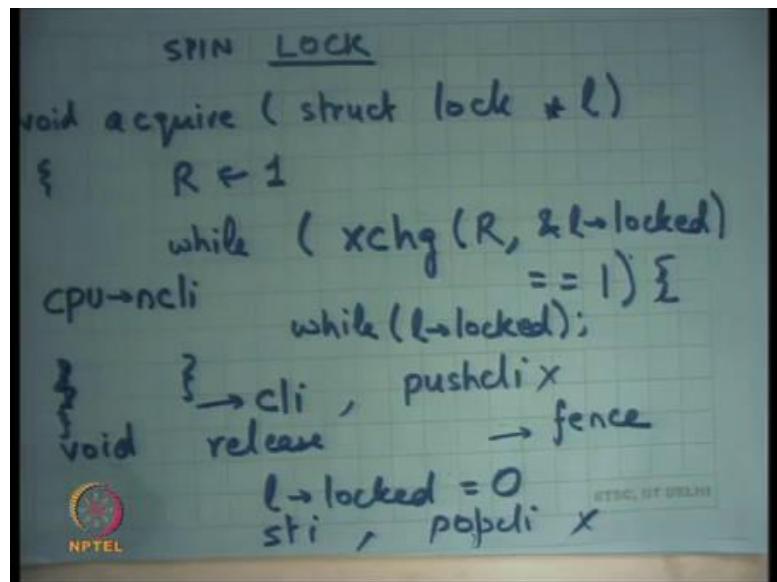
So, what has happened is basically because of fine grained locking any function that is building upon these so, earlier it was very modular, you know move function could have been written in 3 lines and without having to worry about what these functions are doing inside. Whether these functions have to take a lock, does not take a lock that is not my business, I just call these functions. But now because I am doing fine grained locking now it is my business to know, what locks are they going to take right.

And in fact, instead of asking them to take them I need to take them on their behalf, and I will need to take them in a certain order right. So, in other words basically what I am saying is locks and modularity are sort of you know so, locks basically hamper modularity, it is a locks are not very friendly to modularity they sort of make your code more complex less modular right.

Earlier you could just say that this function is going to do delete, this function is going to do add, I do not care what it does internally, but now you have to worry about this function is actually going to need to take a lock, and this function is going to need to take a lock, and because I need to do this atomically instead of them taking a lock let me take a lock on behalf of them. And, now because I am taking a lock, they should not be taking a look and so on right.

So, the entire semantics of your function has become complicated. These semantics are not just that this function this function is going to delete it a name from the directory, the fine semantics now need to be this function is going to delete a name from the directory, and it should not it should assume that a lock has already been taken and it should not be taking a lock it itself right. So, I mean locking and fine-grained locking especially sort of complicates things right. So, let us look at locks and locks implementations in a little more depth alright.

(Refer Slide Time: 10:28)



So, we will we said that our locks implemented you know one of those one of the; one of the ways we implemented locks in couple of lectures back was a spin lock, and where we said that there is a function called acquire right, and the let us take struct lock star 1 let us say and it just says while and let us say this is a function which is internally calling the exchange instruction.

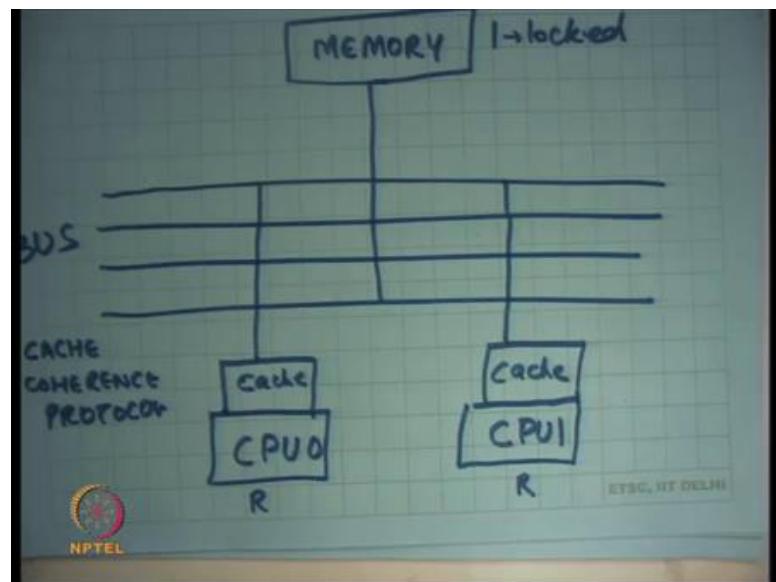
So, if I am calling the exchange instruction. And, so you know one way to do this is let us say there is a register, which I have put a value 1 into and then I say while exchange register address of the locked field in 1 is equal to 1 I keep spinning otherwise I (Refer Time: 11:19) right. So, you know just read this code once more, basically what I am doing is I am basically trying to put the value 1 into the locked field of this 1 right. So, I basically want, I want to put a value 1 into the locked field of the 1 variable except that I want to make sure that earlier it was 0.

Then if it was early as 1 then I should be just waiting for it to become 0 right. So, that is basically acquire that is the semantics acquire and this how I am implementing it and we seen it before. So, I put the 1 value in R and this function is going to atomically swap R and this memory location 1 dot locked right. And, so if 1 dot locked was 0, R is going to become 0 and so, you going to come out of the loop, but if 1 dot locked was 1 then R is going to remain 1 and you are going to retry, but making it you know retry it till you see a 0 value and locked right.

And, we also talked about last time why this implementation is an atomic or you know it works because if 2 instruction 2 threads tried to call exchange simultaneously one of them will occur before the other, they cannot get intently. So, this swapping operation is that all basically right. So, everybody remembers this right all right.

So, let us see what happens at the hardware level then you execute something like this alright. And just for completeness let me just also write it release; release is just 1 dot lock is equal to 0 ok all right. So, let us see what is happening at the hardware level.

(Refer Slide Time: 13:02)



So, let us say here is my bus right. So, we have seen this diagram before, I basically always draw a bus here and I say that here is my CPU right. And, let us say the CPU 0 and this is CPU 1 right and let us say this is memory ok. And, inside the memory there is this variable called 1 dot lock.

Student: (Refer Time: 13:37)

Right. And, in the CPU 0 there are private registers Rs right. And, what I am what each let us say both the threads are executing simultaneously on CPU 0 and CPU 1, this thread is going to set it to 1, this set is going to set it to 1 both are going to say exchange one of them is going to win, whoever wins gets a lock the other one just spins, that is what is the idea. Typically, you have must have studied in your operating system class or in a computer operating class that every CPU also has a cache right. So, let me just say cache.

So, my first question is when I call the exchange instruction is it to just exchange from within the cache? So, I dot locked is just another memory location right and so, when you access it, it just comes into the cache. And, can they exchange the instruction just you know do the local operation without having to go down under disk on the bus?

Student: No.

No because you know because 1, because the exchange operation is an atomic operation and they need and there needs to be serialization between who is doing this you know. So, there has to be some communication on the bus either the communication has to be directly with the main memory, or they have to talk with each other. To basically make sure that you know there is serialization either heavens or heavens right.

So, one of them is going to get 0, another one is going to get the answer 1, both of them cannot get the answer 0 basically. And, so there has to be some bus protocol here that has to happen here and so, each exchange instruction will require some bus transaction right.

In general memory accesses do not necessarily require bus transaction right, whenever I read right a value if the value is found in the cache, I can just locally satisfy it from the cache. It is only when there is a cache miss, I need to go to the memory right. And, typically you know these processors has what is called a cache coherence protocol.

So, the idea is that let us say I access the memory location a and it gets cached here and then this CPU accesses the memory location a then you know there is some protocol that is going on here which will invalidate this location and then valid it and then bring it here right. So, you know if these both the CPUs are accessing the same location then, there will be some bus transactions that are shuttling this variable between these two right.

In any case you know when we are doing this exchange business then the problem is that it is there is a lot of bus traffic basically going on. You know if there are 2 CPUs there is a certain amount of bus traffic, if there are 4 CPUs there both there is more their 8 CPUs even more if the 64 then you know basically bus is definitely the bottleneck. So, cache coherence protocol is for every memory access all right.

So, for every memory access clearly, I mean you cannot have so, the hardware ensures that you know there is some sort of so, there is that is what coherence means. So, there is

coherence in accesses, it cannot be that the same location has two values basically at the same time. So, for every memory access the cache coherence protocol works. It need not work or if the same CPU access at the same location 10 times, it is only the first time that there will be a bus transaction.

The next 9 times it will get satisfied from the local cache, without any bus transaction, without any cache coherence protocol getting having to kick in because assuming that this other CPU is not accessing that location that location is locally satisfied from the cache all right. But if you are executing the exchange instruction each time then you have to make a bus transaction because it has to be atomic with respect to everything else right.

So, in our in the code here; let us say if there are you know if there are; if there are 4 2 processors; one of the processors gets the lock, the other processor just keeps calling exchange and the exchange all each exchange execution exchange is causing a bus transaction and so, there is a lot of bus traffic ok. So, this is not the best possible implementation of a spin lock. And how can you make it better well one way to make it better is for example, put another loop here which is not using an atomic exchange operation, which is just checking.

So, exchange of instruction is a more cost is costly operation because you know it needs atomicity. On the other hand, this operation is just a read operation; a read operation is a less costly operation it does not need an atomicity right. And, so what I am doing is I am basically you know instead of every time calling the expensive operation, where I am basically doing is I am I want to wait; I want to wait for lock to become 0 right. And, so instead of doing it in, but I am doing I also need the exchange interaction, because I want to sort of swap it atomically.

So, that checking code can be done through just reading and then once you are, once the read says yes it has become 0, then you can retry the exchange operation. It is not necessary the exchange operation will succeed, but it is a high likelihood that will succeed this time right. If it does not succeed no problem, you again come back here, and you again wait for it to become 0 right. So, what will happen in this case?

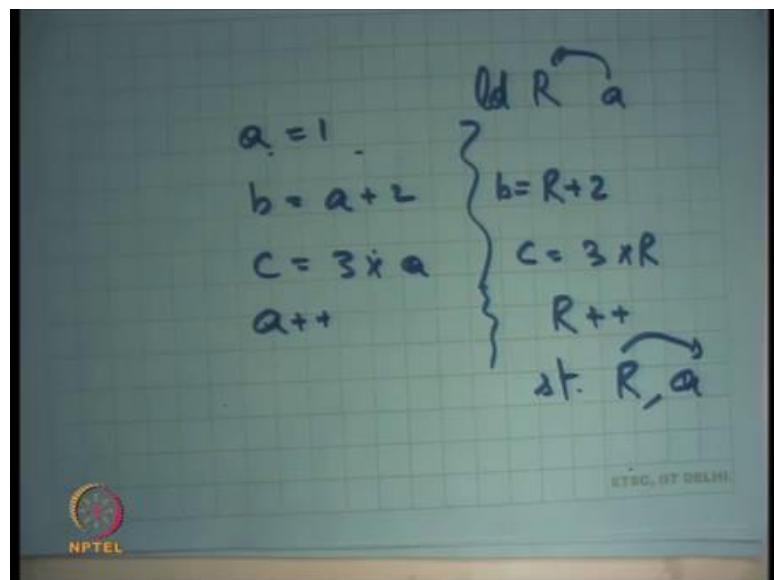
Let us say you know both CPUs try to do exchange one of them wins, the other one just calls the loop and this time that will the inner loop is going to get satisfied from the

cache right. So, the inner loop is going to get transferred from the cache you have reduced the bus traffic all right all right.

So, this is all good, but let us see what happens, if you write code like this you know without having to you know let us say you write this code in C, you just say while exchange and then in the inner loop is while locked. You know if you know a compiler is basically looks at these variables and decides which of these variables to register allocate and which of these variables to keep in memory right.

So, what happens if this variable becomes register allocated right? So, I hope people understand what is register allocation of a variable studied in the programming languages class or right?

(Refer Slide Time: 20:05)



So, basically the idea is let us say; let us say there is a variable called a , and say $a = 1$, you know $b = a + 2$, and $c = 2 \times a$ or whatever. And, so the question is one way to deal with a is basically say that keep it in memory and each of these operations are memory accesses. And, there is other way is basically read a into a register.

So, let us say this is a load instruction and you read a into a register and then you perform all these operations on R . So, you say you know $R + 2 = b$ and $c = 3 * R$ and let us say you also say $a++$. So, you say $R++$ and then later on you can say store R to a right.

So, this is a common optimization a very most basic optimization of a compiler, that if there is a memory it is as a variable instead of so, variables are basically you know have a one to one relation with the memory location, but if there are multiple access to a variable, and the program and the compiler can see that there multiple access to the variable. The optimization is that you just bring the variable from memory into a register do those accesses to the register instead of the memory so, you have saved some memory accesses.

And, then after you have computed at the thing you just save it back into the memory right, common optimization of a compiler. Similarly, in this code this variable l.lock a compiler is free to register allocate. So, what can happen if the variable gets register allocated?

Student: (Refer Time: 21:38).

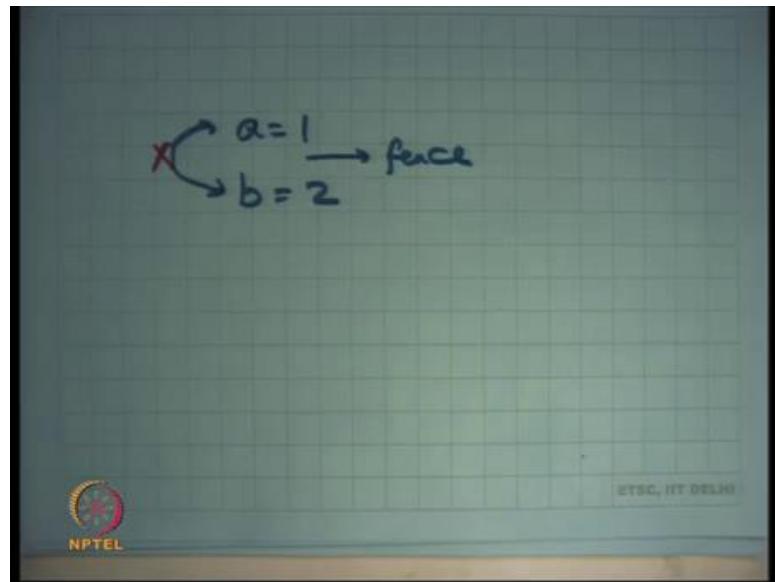
It is an infinite loop right it will never finish. So, you know with the best of the intention compilers are not really you know playing well with what the operating system designer really wants. And, so you know either the operating system designer writes this loop in assembly or actually the compilers give you special keywords to basically say do not optimize this variable all right.

So, there is a quick for example, and see there is a variable called there is a keyword called volatile. So, if you declare a variable or a you know field with a volatile struct or with the volatile type. Then basically the compiler says this is something that you know the programmer has really written very carefully I should not be optimizing it at all right. So, you know just an interesting example of how you know, how a compiler writer. So, a compiler writer does not worry about concurrency and does not need does not understand which one what is the lock and what is not a lock etcetera.

He is just looking at code and he is just you know optimizing it, but you know if you are writing the special code like this you should basically declare things as volatile. And, this is one of the reasons why you know it is right difficult to get concurrent programs correct. Notice that you know it is easy to basically say that acquire right this acquire function very carefully.

And, then use this acquire function to mark critical sections, but on the other hand if I did not want to use locks and I just wanted to very carefully write this sort of code then I have to worry about over the compiler should not optimize it and you know other things like that. And, so that is a very hard thing to reason about in general all right. The other thing a compiler and even in the hardware does is reordering right.

(Refer Slide Time: 23:21)



So, if I basically have an instruction it says a is equal to 1, and then I say b is equal to 2; a compiler is free to reorder these instructions right. For a compiler these are completely different memory accesses completely different variables, it does not matter which occurs first right.

On the other hand if you look at our locking code you know reordering is fatal for our logic, you know because if we are writing to the lock field and then we are accessing some shared variable, if the compiler reorder these things, then you know the critical section is outside the lock or before the lock and bad things can happen right. So, similarly it is possible that the release the critic an access in the critical section is reordered after the release right.

So, for example, in this case 1 dot locked is equal to 0, before that is I had a shared variable access you know these two come are completely independent memory accesses and you know a compiler may say let us just reorder these things. It is not just the compiler who can do this; it is actually even the hardware that can do this right. So, most

so modern hardware basically does out of order memory accesses right even the Intel architecture and most of the performance they get are basically because of out of memory accesses.

And the reason you need to do out of auto memory access is because some memory access is going to take a long time, and others are going to take a short time, because some memory accesses maybe cache hits and others maybe cache misses. So, if whatever the cache hit you know let us just do that first and what does the cache miss do not let it come whenever it when it is ready right. So, even the hard even if the compiler played well with you the hardware can actually reorder accesses.

So, it is possible that the locked access locked variable was in cache and just sort of got you know get what said first and later on the other critical sheared variables getting set. So, once again you have to be very careful in doing this and so, you know modern processors provide what are called fences right.

So, we basically put a fence and the fence is basically saying that all memory accesses before the fence should have finished before any memory access of the after the fence starts ok. So, the idea you know from a hardware designer stand point is that in general let us allow the ordering of memory accesses, reordering of unrelated memory accesses of course, which seem unrelated at least, but a programmer has a way of saying that here is a memory access and here is the memory access.

So, in this case if I want to disallow this so let us say I want to say that this is this should not be possible, then I will put a fence in the middle, so, that is you know. So, there are multiple ways of putting a fence it is very architecture specific you know you have special instructions which you can say that you know his fence there is a fence instruction.

So, you can put a fence instruction. So, that way this will get disallowed or there are special instructions like the exchange instruction it itself acts as a fence right. So, some instructions will never allow reordering of across them themselves all right good no all right.

So, let us look at this implementation again. So, what I am saying is that the exchange instruction itself is acting as a fence in the case of acquire. And, in the case of release the

programmer should put a fence in some way or the other right either a fence instruction or instead of using a simple right you will use some exchange instruction to do the right for example, all right ok.

Now, let us say, so this is a spin lock right and let us say I am an operating system developer and I basically also get interrupts. So, these spin locks will protect against multiple concurrent access by multiple CPUs, but if I am within let us say I am within in within the critical section and an interrupt comes the interrupt handler will get to run. And, if the interrupt handler also needs the same lock then there are problems right, you can either end up with a deadlock or.

So, if you are doing it like this then if I am within a critical section, and I am holding a lock, and it is possible that the interrupt handler also wants to get the same lock then, I will have a deadlock right. So, and you know the most the core of the operating system typically has such code. For example, there is a lock to protect the process table p table in xv6 for example.

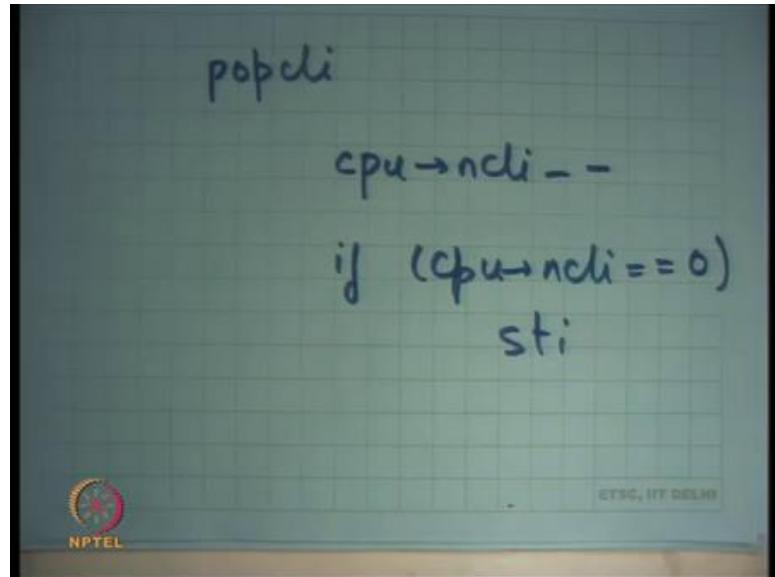
So, that lock is you know is being accessed by multiple functions and even the interrupt handler the timer interrupt handler is going to need to access this p table lock right. And, it is going to need to access the p table and we are going to need to acquire the p table lock. So, such locks are you know or even more special. And, so what you do is in that case you basically make sure that not only do you just do this you also disable interrupts in your acquire and you are enabling interrupts in release right.

So, you know when you acquire a lock any lock, if you know that these locks can be acquired by or can be requested by interrupt handlers you also disable the interrupts. So, within the critical section and interrupts is not possible anymore right. It is only when you really you quit the secretarial section will an interrupt get in the way and. So, you know one xv6 you will find a function called instead of just cli and sti you will find push cli, and instead of sti you will find popcli, and the idea here is that it is possible that you know you are trying to acquire multiple locks.

So, let us say you acquire p table lock first and then you acquire file system locks second and both of them wanted to you know both of them need to do cli, but then let us say you release one of those locks then you do not want to immediately do sti. So, basically you

have some kind of recursion, so each CPU. So, there is a CPU pointer dot so, there is a there is a CPU.ncli variable.

(Refer Slide Time: 29:28)



And, so push cli just does `cpu.ncli++` and if `cpuncli = 1`, then you actually call cli right. Which means you just transition from 0 to 1, so, you actually need to disable interrupts and similarly pop cli.

So, that is push cli roughly speaking and that is pop cli. So, pop cli is basically cpu pointer on `ncli--` and if `cpu.ncli = 0` then `sti` right right. So, clearly, I am talking about within the operating system, where the interrupt handler can require would need to acquire the same lock that you are holding right, only in that case do you need to disable interrupts. And, I am really talking about that the real inner core of the kernel ok.

So, for example, then you see implementation of the spin lock in the xv 6 kernel, and the spin lock is basically used for your p table lock, and among other things you would basically find that the acquire function not just does the exchange to protect against other cpus it also does a cli to protect against interrupt handlers ok. So, I need to do both these things alright. So, and also this ncli variable is a cpu private variable all right. So, there are ways to say that this variable is only going to be accessed by this cpu.

And, so no other cpu can will ever be able to access that value, or you can just have an array with you know first where each element is accessed by only the corresponding cpu and nobody else so, that is a portal per cpu variable.

(Refer Slide Time: 31:17)



So, let us say I am in the user mode ok. So, I have talked about kernel mode, but let us say I am there is a user mode. and I want to do I want to implement my let us a banking application and I want implement locks. So, what kind of locks should I use? Well firstly, question could be you know whether I am running on a multiprocessor or a uniprocessor? Or in fact, even before that the question should be whether you want to implement a spin lock or a blocking lock, right?

So, if you want to do a spin lock do you need any kernel involvement unless you want to disable interrupts? What a user level lock needs to disable interrupts? I said you need to disable interrupts only if that lock could be requested by an interrupt handler. I mean assuming that the user level locks are just private to the user and the kernel has nothing to do with it, then the interrupt handler has nothing to do with that lock right. So, you will not need to disable any interrupts for a user level lock ok. So, can you do implement a spin lock without having any kernel involvement? The answer is yes ok.

All you need to do is declare a variable and use the exchange instruction; exchange instruction is an unprivileged instruction right; it just has the semantics that things will be atomic that is all right. So, the same code that I showed you this one without the cli

implements a spin lock in user mode ok. So, a spin lock in user mode is as fast as a spin lock in kernel mode you just basically you know try to atomically set it to 1, and if not you just spin just in exactly in the same way, and hopefully your critical section was small and you will immediately get the lock all right.

Does it matter whether you are using kernel level threads or user level threads, because user level threads will only run on a single CPU you do not even need to do this exchange business right. User level threads will only run on a single CPU and so, instead of using a spin lock you would probably want to use a blocking lock instead right.

And of course, so, blocking locks will be used either, if you are using user level threads or if you are sure that you know your threads are not going to run on a single CPU for whatever other reason there could be and if your critical sections are known to be very large right. For example, if you are making a system call while holding that lock you might as well just you know use a blocking lock, rather than using a spin lock right.

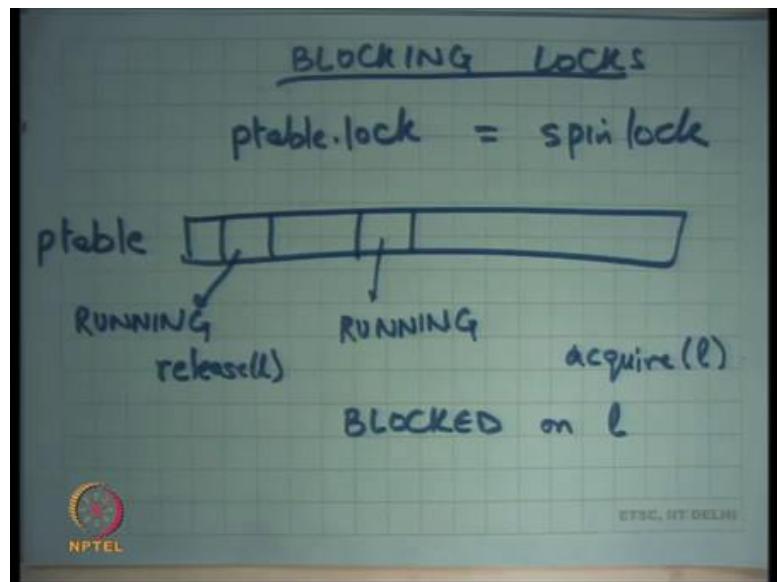
So, in all these cases you will not even spin lock you will use a blocking lock. Do you need kernel involvement to do blocking locks to implement blocking locks?

Student: Yes.

Yes, because a blocking lock basically needs to tell the kernel to change my state from ready or running to blocked right. And, so I need and the only the user has no way of changing it from ready to block and so, it has to tell the kernel to do it right. So, there has to be a kernel interaction unless of course, you were using user level threads in which case the kernel has no idea and so, you are in that case your user level scheduler is just going to is just changing the state of your you know a currently running thread to from ready to block.

So, in that case you are p table is maintained at the user level. So, in either case the p tables, the state in the p table needs to be changed from ready to be blocked. If you are running kernel level threads you need kernel interaction to do that if you are running user level threads you can just do that locally in the user all right. So, let us see how blocking locks are implemented right.

(Refer Slide Time: 34:48)



So, you can imagine that there is a p table right or you know I am using a an array, but you could even have a list of PCBs, or any such data structure that is maintaining all your process PCBs, Process Control Blocks right. And, what you are going to do is let us say somebody says lock and he is not able to get the lock then you will basically want to change its state.

So, let us say this is a process and this is currently running, then and it calls acquire you would want to change it is state to blocked right. And, you will want to record that it is blocked on whatever was argument of l acquire, so let us say blocked on l right.

And, then if somebody calls some other process so, this becomes blocked. So, this never gets to run in future till somebody calls release. So, let us say here is the process that was running and then it calls release l. And, what release is going to do is it is going to go over the p table right and pick up one process, that is blocked on l right. So, this from here this here the ls are matched and change it from block to running already not running, but ready it will change it from block to ready all right. So, that is how blocking locks will be implemented all right.

So, but this p table structure itself it needs to be protected, you know accesses to the p table structure it says by multiple threads needs to be protected. So, you will use a spin lock to protect the p table and then and so, blocking lock internally will use a spin lock to protect this structure and to switch from running between running and blocked these

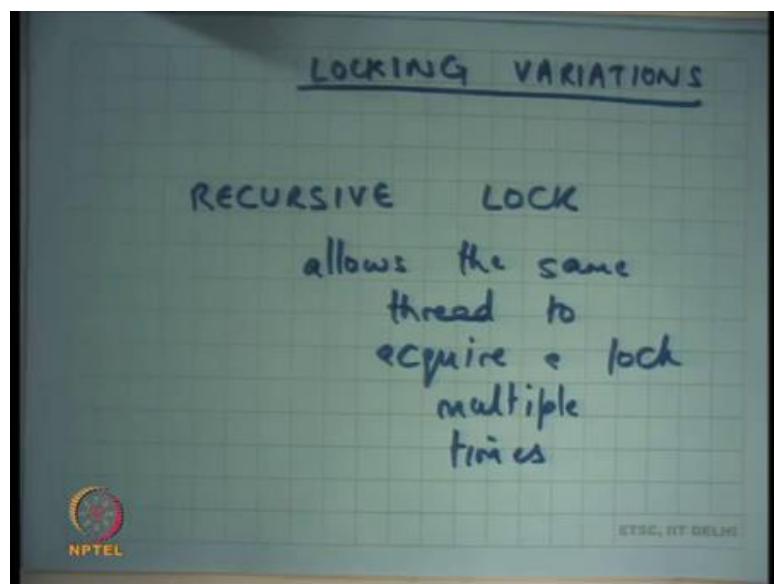
different entries right. So, there will be as p table dot lock let us say which will be a spin lock.

Student: P table dop lock (Refer Time: 37:06).

Well I mean will the p table dot lock only be needed for a multiprocessor, well you know. So, on a uniprocessor a p table dot lock equates to a cli no clear interrupts. So, basically you want that while you are in the middle of accessing the p table nobody else should basically interrupt you right.

So, on a multiprocessor you will use a spin lock on a uniprocessor you could do that just by disabling interrupts all right. Basically, what you want is mutual exclusion why you are accessing the p table right. And, mutual exclusion on multiprocessor only base spin locks mutual exclusion on a uniprocessor the way is disabling interrupts.

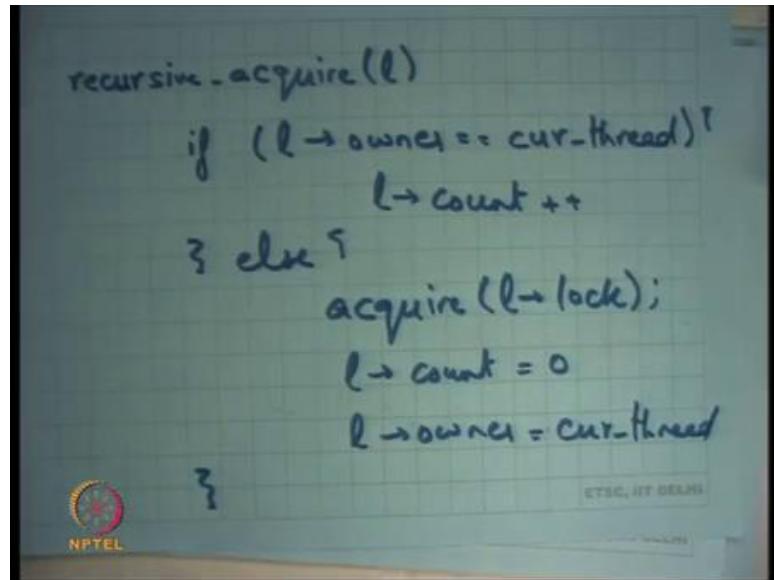
(Refer Slide Time: 37:49)



Now, let me talk about some locking variations all right. So, there is something called a recursive lock. So, you may have seen that sometimes we run into this situation where, you acquire a lock and then you call some other function and that once you acquire the same lock, and at that point we deadlock right, because the same thread cannot acquire the same lock multiple times.

So, you know the recursive lock basically what it does it allows the same thread to acquire a lock multiple times all right and the semantics of a recursive lock up fairly simple.

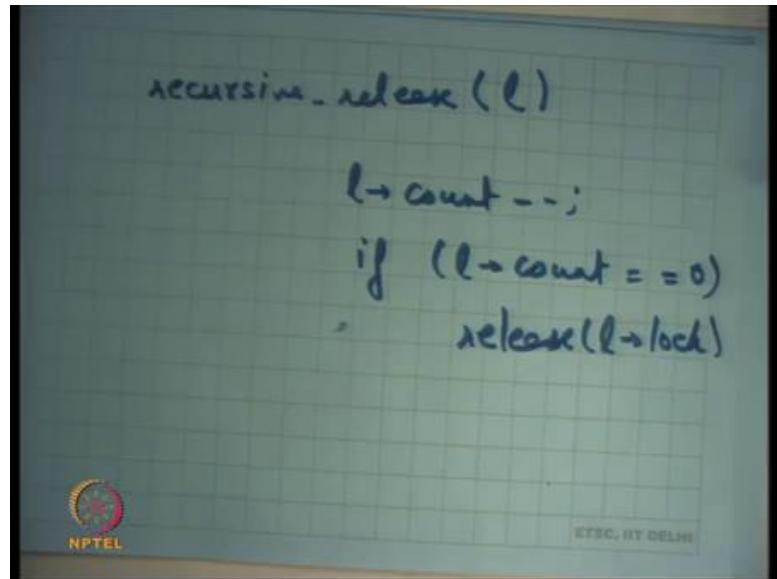
(Refer Slide Time: 38:52)



Let us say this is recursive lock; let us say this is recursive acquire l, you will say if l.owner you will keep something called an owner is equal to current thread, then l.count++ all right else you call the regular acquire all right. And, you set l.count to 0 and l.owner to cur thread right.

So, basically the idea is you know a lock is supposed to provide mutual exclusion between multiple threads. If for some reason the programmer feels that you know or for modularity or whatever reason, if he feels that the same thread wants to acquire the same lock multiple times let us allow that right. So, that is a that is the; that is the spirit behind a recursive lock. And, of course, release will just basically decrement count and only if count becomes 0 does it release the lock right. So, that is what relieves me does. So, should I write release?

(Refer Slide Time: 40:13)



So, let us say let me also write release, I will just say `l.count--` if `l.count = 0`, then release `l.lock` right something like this right. So, this is a recursive lock sounds like a good idea or a bad idea? No idea ok.

Student: [laughter].

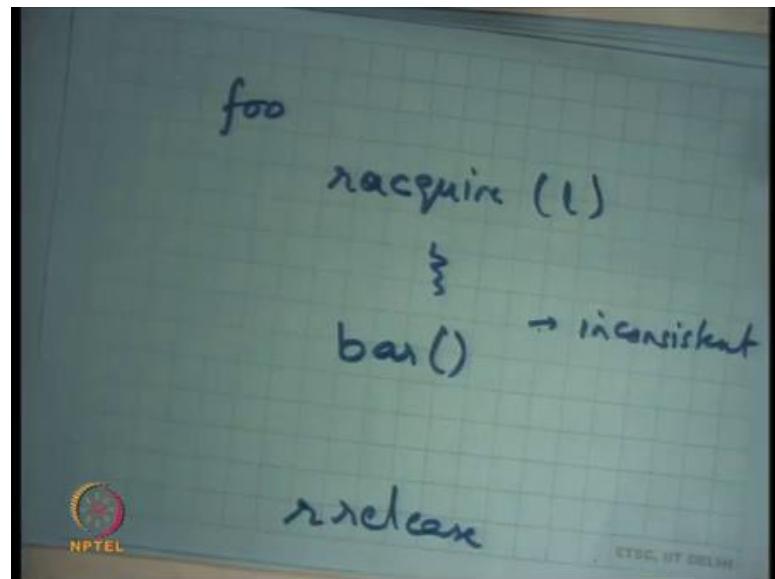
So, it is actually a bad it is generally considered a bad idea to do a recursive lock all right and why. Basically, usually the semantics of a lock is that when you acquire a lock you know at the point, when you require the lock and you just enter the critical section you can pretty much assume that this that the state is it there is a consistent state of the system right.

So, if the idea is that if you have been able to acquire the lock, anybody else who has released the lock has left this has left the state in the sheared state in a consistent state right, where is less that left the memory in a consistent state right. That is basically; that is basically have that is basically been are invariant right that if I am able to acquire the lock I can assume that at the first instruction of my critical section, the system is in a consistent state.

And that other invariant I usually maintain is that just before I release the lock, I have ensured that the system is again in consistent state right. And, so then I release the lock, so, that the other person who acquires it will also see the system in a consistent rate. So,

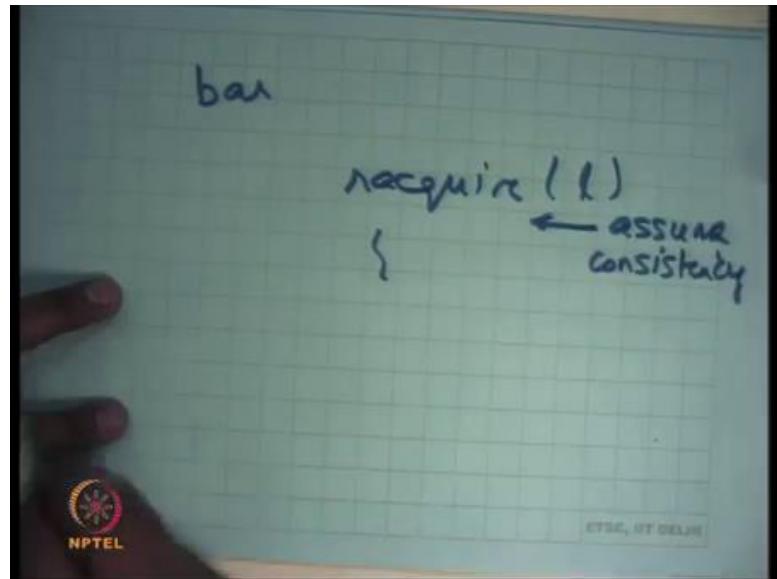
generally you know you the assumption is that as you have acquired if you have acquired the lock the system is one consistent strait and you will maintain it in a consistent state before you release the lock or you will keep it in the system.

(Refer Slide Time: 42:15)



But, if you do recursive acquire then you know then it is possible that you have a function foo that you know says let us say I am going to say recursive acquires r acquire l, does something makes it inconsistent right make the state in inconsistent has not released the lock right yet. So, he is going to say r release here somewhere here, but he calls bar here right.

(Refer Slide Time: 42:37)

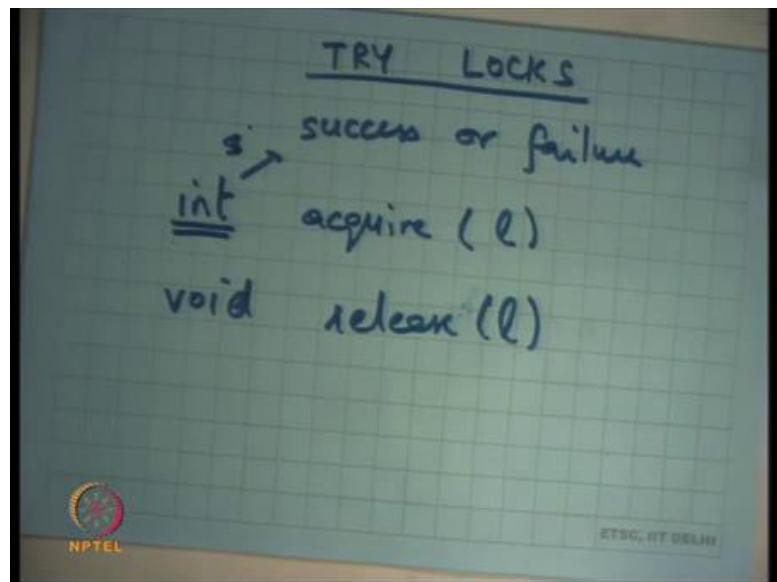


And, bar internally is going to say r acquire and he is going to start doing something, but he is going to assume you know assuming that these code has been written you know in modern modular fashion in a different file or different program or whatever he is going to probably assume that it is inconsistent state, for assumed consistency. But because you know you are using recursive locks you will know you will violate that assumption and this bug will be much harder to find.

So, in fact, you know using recursive lock, you have made it easy for your program to have bugs right, that have not been that cannot be found. On the other hand if you did not use the recursive lock, you know the first call to bar would have told you over there is a bug in a program right, because there would have been a deadlock right there right.

So, in general you know a programmer wants to keep his thinking simple and consistent with this idea that when you get a lock, things are consistent when you release the lock things are consistent. And, if the programmer is indeed doing that then recursive acquire is a bad idea all right ok.

(Refer Slide Time: 43:44)



Then there is another variation of locks called try locks ok. So, what are try locks? Instead of so, the it is the same thing let us say instead that so, the idea is that acquire 1 has a return value now int right, which basically says and you know the release is just void and the acquire basically says success or failure right.

So, in our regular lock and acquire basically always succeeds or it waits. In the case of a try lock you try to get the lock. If you did not get it you just return a minus 1 or you know a failure value right and when you and so, it is up to the caller to do whatever he likes of course, you know you can implement a regular lock using a try lock very easy.

You can just sort of put the try lock in a loop and you get a you get a regular lock; it may not be the most efficient way to do a regular lock right. But the advantage of a try lock is that gives some flexibility to the caller, he may want to say let us try to acquire this lock, if I do not get it then I have something else to do let us do that first right and then retry it. So, it gives him that flexibility.

On the other hand, and the previous lock and acquire basically is committing that I am definitely going to I am going to either do that or wait basically right. So, try lock gives you some flexibility into you know whether you want to whether you want to wait or whether you want to and do something else.

Let me now discuss a real example. So, I hope you all know that the banking example that I took earlier was a very very hypothetical example for many reasons. Firstly, you know bank accounts are not maintained in memories, secondly, usually do not write code in such a way where you are going to do a global sum operation on all the accounts, you would want to do some kind of more distributed and segmented way of calculating sum.

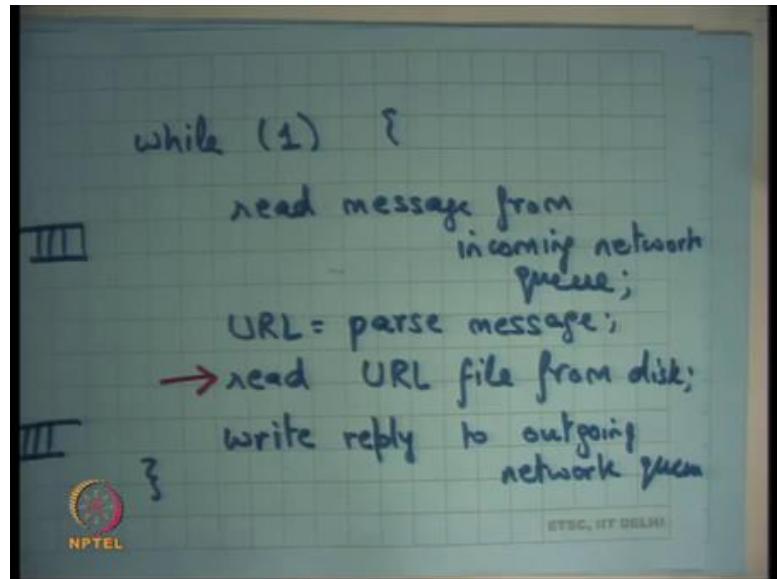
And, so that there is more scalability in your system or you know whether you want to calculate some at all you can just you know update the sum as the transfer is going go on or something like that. In any case it was just an idea a way of telling you know what the problems of finding and locking are.

(Refer Slide Time: 46:05)



Let us take a more real a real example of a web server alright. So, what is the web server? Web server is let us say you know this running on this machine, which has a disk and it has a network ok. And, a client sends an HTTP request and receives a reply, HTTP response. And, tip I mean let us here let us take a simple case where the HTTP request is a URL and the reply are the contents of that URL, which is an HTML page let us say right.

(Refer Slide Time: 47:13)



So, how is the web server like this implemented? Well let us say, you know at a very high level the web server is probably running a loop like this while 1, so, while true know it is an infinite loop read message from incoming network queue.

Let us say URL is equal to parse message all right, read URLs file. So, whatever is the URL you know you can parse it to forget a file. So, let us say read the URL file from disk right and then right. So, you get the URL file from disk you get the contents of the file and then you write those contents also right as a reply so, you write the reply to outgoing network queue right. So, what am I assuming here I am basically assuming that there is a network queue, right?

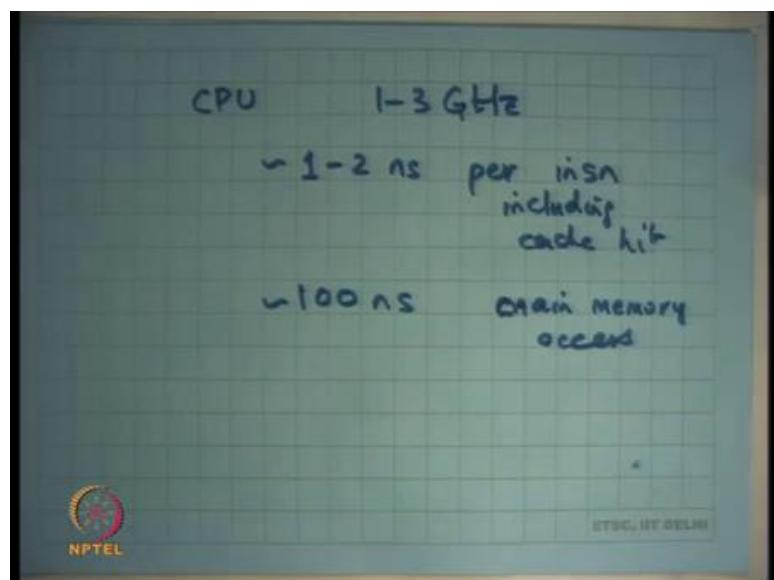
There is somebody who is filling up this network queue. So, there are packets being received on the wire and, those packets are getting stuffed into this network queue incoming network queue. There is this server that is running that is picking up packets from this incoming network queue, processing them in this way and then there is an outgoing network queue, which you just and the server is putting things with an outgoing network queue and there is somebody was paying things up from the neck out doing network queue and putting them on wire right.

So, let us see, what is the performance of this web server all right. So, basically what will happen is let us say the multiple clients in this let us say they are you know there are multiple clients, that accessing this web server their request will get queued in the

incoming queue, and the you know depending on how many clients there are what is the concurrency level of clients the queue will keep filling up and this server will pick up one request and start serving it. So, you know the maximum number of clients that it can serve in a second is depends on how much time it takes to execute this code right.

And, how much time does it take to execute this code? By far the most expensive operation in this is this right. Reading the URL from disk is by far the most expensive operation. These operations are likely to finish in you know 100s of nanoseconds to may be microseconds or something, but this operation URL from disk is an operation that takes milliseconds to complete right. Why does disk take so much time, while the other things are so much faster? Have you discussed this before? No ok.

(Refer Slide Time: 50:13)

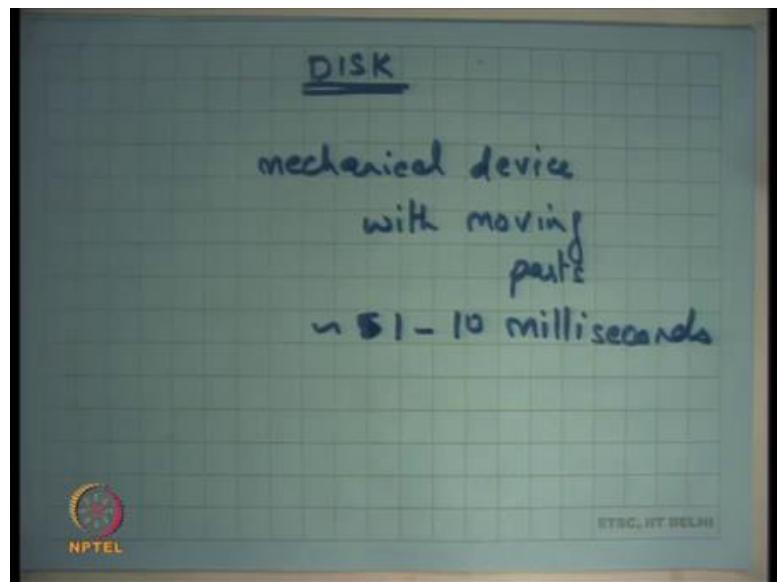


So, let us say there is a so, typically today's modern CPU runs at 1 2 you know, let us say 3 giga Hertz right or roughly 1 to 2 nanoseconds per instruction so, 1 to 2 nanoseconds per instruction. If the instruction was a memory access and the memory access is a cache hit, then also you know typical execution times are 1 2 3 nanoseconds.

So, cache hit including cache hit, if it is a cache miss then you know let me put approximately and roughly 100 nanoseconds for a cache miss or let us say main memory access right.

These are all electronic operations these are just you know semiconductors exchanging electrons to basically access either cache or memory or things like that, the only reason memory is sort of more costly is because you have to travel a longer distance. You have to go over the bus there is some bus contention that you to worry about and then you get to the memory and then you know, but it is all electrons traveling and so, it is very fast right. On the other hand, you know a disk access or a magnetic disk which has persistence is a mechanical device right.

(Refer Slide Time: 51:35)



So, a disk actually if you look at a disk, then it is a mechanical device so, it is a mechanical device with moving parts ok. Exactly what is the structure of a disk and why and so, you know hence it is much more costly and it is on the order of you know 5 or let us say you know 1 to 10 milliseconds ok.

So, that is 10 to the power a million times slower than an instruction access, it is a or which means that accessing a disk operation in that time you could actually have executed a million instructions in CPU. So, we going to discuss how our disk is organized exactly and what determines whether what the access time is exactly. And, then what does it mean for a web server and it is scalability which means how many concurrent clients can it support and how you can optimize it and what role does multi-threading have to play in optimizing it all right. So, we are going to look at that ok.

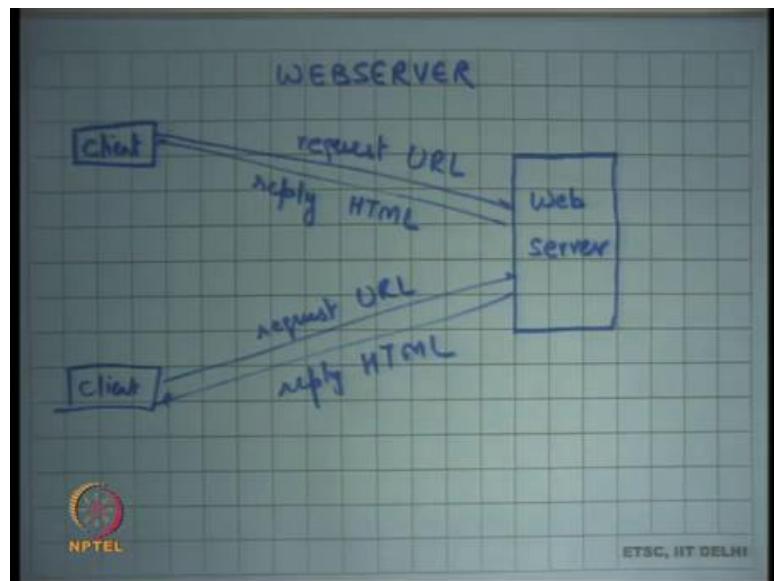
Thanks.

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

Lecture – 24
Condition variables

Welcome to Operating Systems lecture 24 right.

(Refer Slide Time: 00:30)



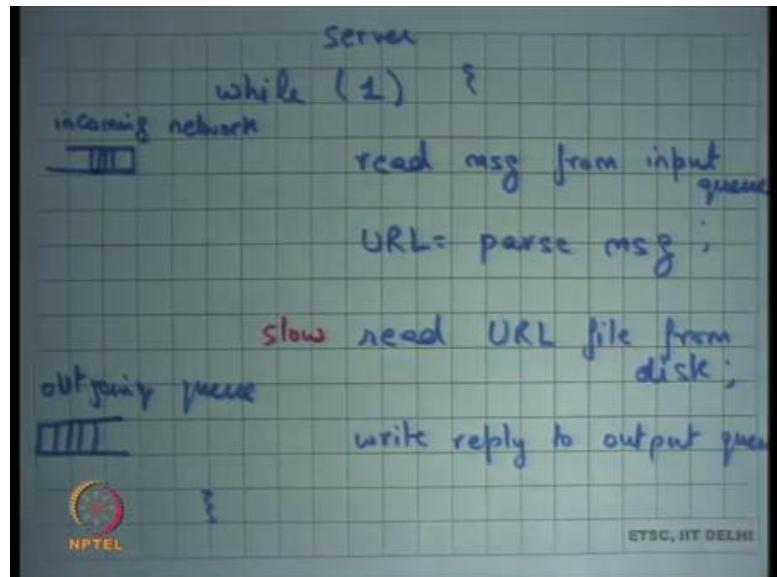
So, last time we were talking about locking, we were talking about fine grained locking, what kind of problems we can run into while doing fine grained locking. And how deciding which locks to use, and where to put the acquire and release statements it is a bit of an art, and one needs to do it.

By looking at the program as a whole and we were looking at this toy program which was maintaining bank accounts etcetera. And I am going to take talk about a more realistic example today which is a web server. And let us say you know web servers implemented on a single machine in our case.

Let us say and there are multiple clients that can connect to this web server. And the clients connect with the network, so client will send a packet which will contain the request which will know for an HTTP server it will be a URL.

And the server will reply with a page which will be the corresponding HTML page right. And multiple clients can connect with this server simultaneously alright.

(Refer Slide Time: 01:28)



So, this is you know the pseudocode of the web server you have an infinite loop. And this infinite loop does just this that it reads messages from an input queue right. So, this is the incoming network queue, right, and parses the message obtains the URL from the message. Reads a file from the disk, which is correspondence to that URL right, and writes the message to an output queue.

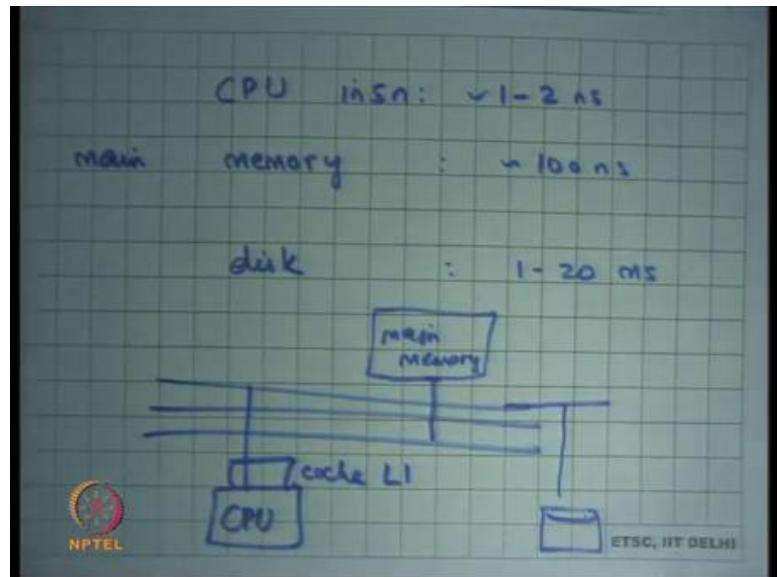
Let us say this is the output queue or outgoing queue, so we just focusing on the server's logic, so this is the server. And you know there is some logic which is basically picking packets off the wire from the network and putting it in the incoming network queue. And then there is some logic which is the network which is picking packet from the outgoing network queue and putting them on the output wire right alright.

And last time we were discussing that in this entire operation you know reading message from an input queue is as fast you just need to copy some data across memory. Parsing the message involves some CPU instructions, reading URL from the disk is expensive because it needs to access the magnetic disk.

And the thing we were discussing last time is that the magnetic disk is a mechanical device as opposed to all these other things which are electronic devices right. And then

you write the reply to the output queue right. So, we said that this one is this step is the slowest of them all and let us look at how slow it is.

(Refer Slide Time: 03:15)



So, we said that the CPU instruction that executes without having to go off chip right. So, when we draw this diagram and I say this is the CPU many instructions. And let us say this is the cache this is you know what is called the L1 cache for a CPU. Let us say and so all this accesses within the L1 cache and to the registers are on chip access, so they do not have to go outside the chip, so these are also called on chip accesses.

And all the on-chip accesses are really fast, and they because you know assuming you are about one to two giga Hertz processor. They will typically take 1 to 2 nano seconds to execute a single instruction assuming it is not going off chip right. Now, main memory on the other hand is off chip right.

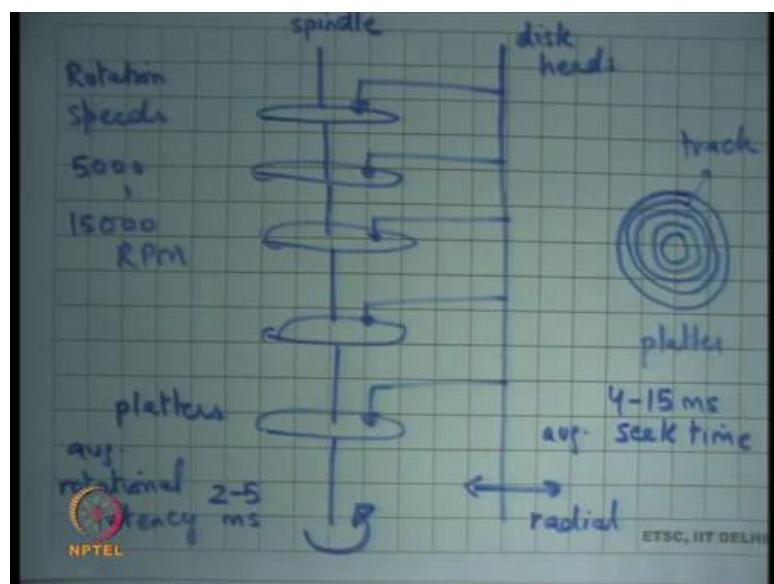
So, this is the main memory and anything which is a cache miss in L1 has to go to the main memory and it has to go through the bus. It has to travel a longer distance, also it has to talk to the bus to make sure that they are you know multiple devices on this bus example multiple CPUs or other things. Then there has to be some contention protocol that has to be followed, and so, it takes longer.

And it takes roughly 100 nanoseconds to access the main memory. On the other hand, saw something like a disk, the disk transaction also goes through this bus. But,

eventually the bottleneck, so let us say I had drawn a disk somewhere here. The bottleneck is really in the disk device itself because the disk device is a mechanical device.

And we said that the disk usually takes on the order of milliseconds as opposed to nanoseconds. So, actually it is you know a million times slower than your CPU right. So, let us just understand a little bit about how a disk is how magnetic disk works right.

(Refer Slide Time: 05:04)



So, a magnetic disk is organized as a cylinder with such platters alright. So, these are this sort of these circular plates what is called platters all right, this axis is called the spindle. And the spindle is the axis of rotation right, so it just rotates right, and it rotates at a high speed.

And then there is what is called a disk head disk heads actually, so the multiple disk heads. And they are these disk heads that sit on top and are able to read information out of these magnetic platters. It is a bit like the gramophone that you may have seen in old movies or you know in antique sort of things. So, it is a similar thing there is a head which sits on top of this magnetic disk and the disk rotates.

And the head basically reads off information off the magnetic disk and the and basically takes this head is rotating and the to allow the magnetic to allow the disk head to go anywhere the disk head has a capability to move radial right. If I were to draw a platter, a

platter internally is just concentric rings, so this is you know another view of the platter, so if I just look from the at the platter from the top a platter looks something like this, and each of these concentric rings is called a track ok.

And, so the disk head moves radially to position itself on a track. And because the spindle is rotating it will eventually be able to reach the position it wanted to reach, and it will be able to read stuff out of the disk all right ok. So, the average, so both these mechanics, so there are two mechanical movements that need to happen for you to access a disk block. The first movement is a radial movement of the disk head itself. And the second movement is the rotational movement which is usually happening at a continuous speed of the spindle. And both these movements define how long it takes to access a disk block right. So, typical rotation speeds so, rotation speeds of a disk are 5000 to 15000 RPM Revolutions Per Minute. So, if you know if you have a laptop, so some kind of mobile hard disk. Then it will be slower it will do 5000 RPM if it is an inter-enterprise grade hard drive then it will have faster. So, typically ranges between 5000 to 15000 RPM which translates.

So, and so the disk and similarly the time it roughly takes for the radial movement to reach the correct track depends on the distance from the edge to the center of this platter what is you know typical seek times roughly range between 4 to 15 milliseconds that is the average seek time. Once again you know depends on the hard disk and the kind of environment setup in if it is an enterprise environment where you know it will be faster.

So, it will be roughly at the level of 4 milliseconds if it is mobile environment then it will be slower it will be 15 milliseconds and so on. So, let us say I make a request to the disk I want I say I want to access disk block number x. Then it will figure out that x lives on this platter on this track and at this offset.

And, so basically it will first position the disk head at that particular track, and because the disk is rotating eventually it will reach x and as soon as it reaches x it is going to start reading off data from there right. And then once it reads off the data it just puts it in some buffer you know there are multiple levels of buffering that is happening that are happening. One buffer is within the disk device itself, so the disk controller which is maintaining it is doing all this logic.

So, there is must be a disk controller that is receiving a request from a computer and then you know implementing all this logic, and that disk controller typically also has a buffer. So, you know all this data that is being read from the magnetic disk is stored in this electronic buffer. You know it is and the buffer has the same technology that you will have for your main memory for example, right. And, so you will basically store it in the main memory and eventually you will send it to the CPU right ok.

So, this is this radial movement, so is also called the seek time right. The time it takes to reach the correct track through the radial movement of the head is also called the seek time of that head. And the time it takes to, so once you are on the head and your from you know once you are on the track the time it takes to actually reach the data you were interested in is called the rotational latency.

So, that is called the rotational latency, rotational latency will depend on you know it is just a matter of chance, if you know as soon as you seek the correct track if it. So, happens that the data you are interested in is right there you will just read it right of it if it. So, happens that it just went by you have to wait for a full revolution before you reach it right.

So, it is just a matter of chance on average the rotational latency will be the time it takes to do half a revolution right. And, so the average rotational latency ranges between 2 to 5 milliseconds alright because that is average. Similarly, I am talking about the average seek time now; once again seek time also depends you know it is also a matter of chance.

You know if you happen if the block you are seeking happens to be very close to your current position then you only have to travel a few small distance and so, seek time will be fast on the other hand if the block happens to be very far then you will have to travel a long distance so, that seek time will be more right.

It is also a matter of optimization often you know as we are going to discuss later in the course. Layout of your file system and layout of your disk should be optimized to minimize the seek time right. Rotational latency you cannot do much about because you know it is just it is a matter of the disk is continuously spinning.

So, you know where you reach when you reach there is sort of completely random. But the seek time you can optimize by making sure that things that are likely to get accessed

together are close to each other. Or the tracks at least either they are on the same track or they are on very close tracks each other alright.

This also means that random accesses to the disk are more expensive than sequential access to the disk right. Let us say I just say I want to access x then y then z you know these are complete and x and y and z are completely random disk numbers. Then it will tell you know there will be a lot of movement of the disk head to be able to get from x to y and then y to z and so on. On the other hand, if you say I want x 2, x plus 100 that is very fast right, because it is slightly to be on the same track.

So, you just say you know you do not pay the seek time again over and over again you only pay the seek time. For the first block you also pay the rotational latency only for the first block all the other 99 blocks you only pay the data transfer time right which is just now going that much distance and data transfer times are actually much smaller typically than the rotational and seek time. So, random accesses are more costlier than sequential accesses and what is the exact trade off we are going to look at when we are going to study the file systems.

Student: Sir is the movement of disk heads independent of each other or.

Its independent of each other yes, the disk heads and although I would draw it like this, I mean typically you will basically have independent disk, and there is a lot of optimization you know. So, these are the you know what I have drawn is a very strict diagram kind of thing, but there is a lot of optimization that goes on in the middle you know the.

So, and this is all the specialty of people who manufacture these disk devices alright. So, with that we basically you know end up with a number something like 1 to 20 milliseconds to do a disk access right, and where does that leave us in terms of a web server.

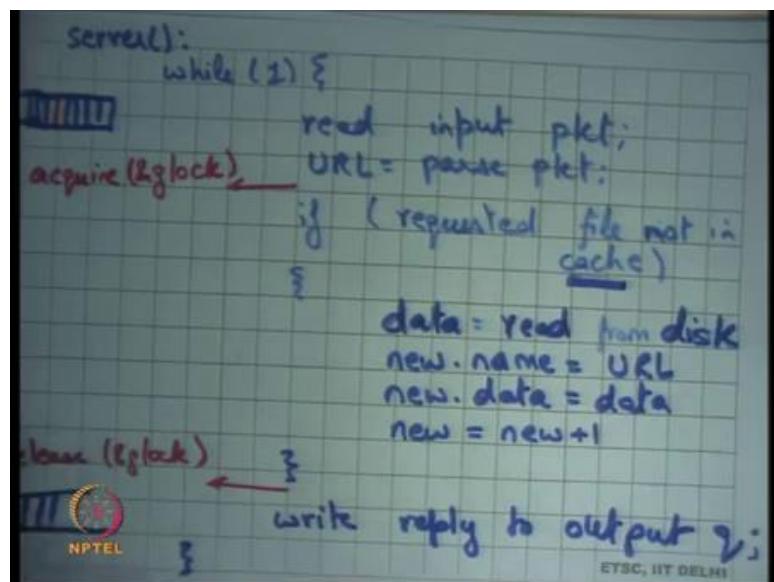
So, in this web server it basically means that every access, every request to the web server will take roughly 1 to 20 milliseconds. Because, you know this is by far the longest slowest step and everything will get serialized here. And you will basically every request will take 1 to 10 20 milliseconds let us say.

And so, how many requests can the web server's service at per second? So, what is the; what is the throughput of the web server 1 upon 10 milliseconds that is roughly 100 requests per second right. So, with this kind of a code you will only be able to service up to 100 requests per second.

So, what will typically happen is let us say you are getting requests which are at a higher rate than 100 seconds. Then they will start they will eventually fill up the incoming queue and packets will start getting dropped at the incoming queue right. And so, you are basically operating at 100 requests per second. 100 requests per second is not a very not a good number at all modern web servers deal with tens of thousands of requests per second alright.

So, what do what is the first optimization that comes to mind, well the first thing is you know why I have to read from the disk every time. It is quite likely that the same pages are being requested by multiple clients which is often the case right. So, once again we are taking the taking this com taking advantage of locality spatial and temporal locality. And we are going to use caching and we are going to cache the contents of the disk in main memory right.

(Refer Slide Time: 15:33)



So, let me rewrite this code for the server and I am going to say while 1. Once again read input packet this short handing it little bit. Let us say URL is equal to parse packet, and then if requested file in cache, then is not in cache is not in cache. Then data is equal to

read from disk, new dot name is equal to URL, new dot data is equal to data alright. And let us say let us have some counter which says new is equal to new plus 1 here all right. And then I just say write reply to output queue right.

So, basically, I have just implemented a memory cache in memory cache, and I will first check if the requested file is not in the cache. Then I will read the data from the disk allocate an entry in the cache and I am sort of assuming that there is a lot of free space in my cache.

So, let us not worry about cache replacement and other things. Let us just say as this point are called new and the new is the current empty slot. So, you just fill up the empty slot with some data and it is called funny URL and you increment the new right. So, this is better because if there are if 90 percent of the requests hit in the cache then 90 percent of the requests can be served from the cache 10 percent of the request need to go to the disk.

However, still if you think about it, you have an input queue right; if even if one of these requests hit misses, so let us say missing one is this red one. So, even if one of these requests misses all the future ones have to wait right. Because, even if one of those misses the thread basically gets busy accessing the disk and waiting on the disk and disk is going to take milliseconds, and so all the future requests are going to wait right.

So, that I mean eventually the throughput of the system does not really improve even if there are a few misses in the system ok. And more importantly the latency the user the perceived latency so, when you; when you; when you type your URL on the browser the user perceived latency becomes very bad. Because anybody who is blocking you know even though you your request could have would have been served very fast. Because there is somebody standing in front of you whose request would take a long time you have to wait for that request to get seek.

Student: We can maintain a new queue just to keep track of just to.

Just to keep track of.

Student: The missed ones (Refer Time: 19:03).

Right, so one way to do this is basically maintain a new queue and basically you say that if requested file not in cache put it in different queue. But then do you need to start the disk and you need to have some thread that is waiting on that disk while you start taking another request right.

So, there is some amount of concurrency that has to happen. Even if you make two queues you also need two threads of control; one thread of control is going to wait on the disk and another thread of control is going to pick up the next packet from the input queue right. And how many threads will you maintain?

So, once again the solution that has been proposed as let us have you know at this point if the requested file is not in the cache. Do not do all this in the context of the current thread. Spawn a new thread right and let that thread wait on the disk and I can go forward and pick up the next request right.

So basically, we are talking about some kind of multithreading and question is how many, so one option is do it you know on demand. So, every time you get something like this spawn a new thread as soon as it finishes you know join it, so that you know terminate it and so, you can you can do it in that way right.

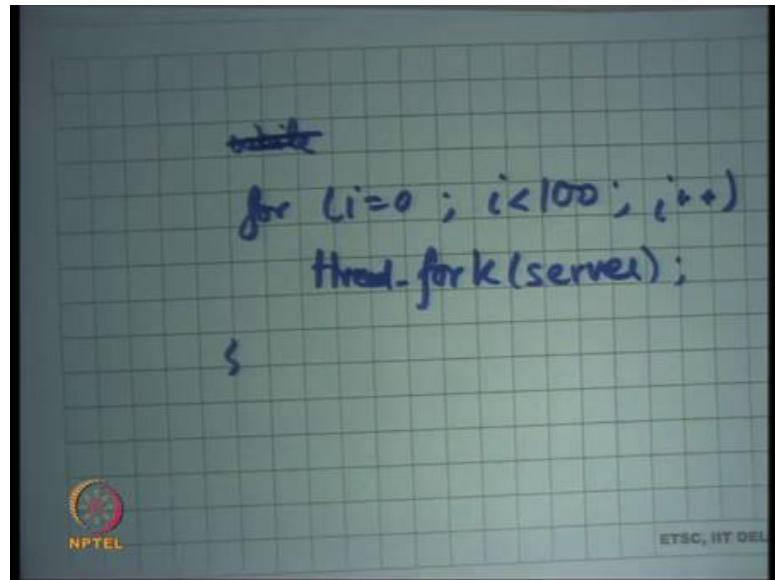
And in either case basically what you are saying is that you need to have multiple threads of control and you need to have concurrency right. So, basically this is an example where you need concurrency to improve your through system utilization and also reduce your response times.

And this is an example where there you need concurrency because there is there are two devices in the system there is a CPU and there is a disk. And you want that both of them should be kept active, and the way to keep both of them active is to have multiple threads one you know few threads is waiting on the disk this is much slower.

So, when many threads can be on the disk simultaneously and let us say one thread waiting on the CPU. Similarly, you know we have seen other examples where the multiple CPU's, so if you want to keep multiple CPU's busy then also you need multiple threads right.

So, in one in say another way you can think of a disk just another processing element, and if you want to use multiple processing elements simultaneously then you need multiple threads all right. So, what are you going to do you are going to say let us say one way to do it is let us call this the server, right?

(Refer Slide Time: 21:28)



So, let us say this is the function, which is the server, and so, you could basically say something like this. You say I am going to spawn a 100, so let us say for i is equals to 0, i is less than hundred i plus plus fork server all right.

So, I forked and let us say this fork is basically a forking a trend or if you prefer let us say thread fork. So, it starts a thread and it basically says that this thread should be running this particular function called server. So, now, there will be going to be 100 threads that are simultaneously executing this function called server.

And now, what will happen is there are 100 threads that are executing the server simultaneously one of them or many of them could be executing on the queue, some of them could be waiting on the disk. And, so you have higher system utilization overall right. How do you choose this number 100? Well if it is too small then you know you can have poor system utilization.

Because, you could have you know you could have multiple threads waiting on the disk simultaneously, and the disk could you know take multiple requests simultaneously. On

the other hand, if it is too large then there is no problem from a concurrency standpoint actually if it is you know if you have infinite threads that is good you know completely logically speaking.

But you will run out of your memory resources to actually manage these threads right each thread takes some memory for example, each thread is the separate stack right. So, threads have some overhead, and so if you have too many threads then you know you are going to soon run out of your memory to manage those threads.

So, there is a tradeoff you basically want the maximum concurrency level without having too much memory over it. And so, you and how and, so you could basically you know one way to do it is just spawn threads on demand. And when you hit your memory limit then stop spawning the threads you know that is one that is another way.

But the all these issues all these decisions basically come in the realm of scheduling, how many threads to spawn which thread to run when etcetera these are basically scheduling issues. And we are going to look at them as when we go talk about it later in the class alright.

So, now I have multiple threads that is accessing this code is there a is that no that is not ok, because there are too many shared resources here right there is a shared so firstly, they are there is a shared input queue multiple threads accessing the same input queue you need synchronization right.

Similarly, there is an output queue here right multiple threads accessing the output queue you need synchronization and none of it is present here and bad things can happen right. Because, it is a shared data multiple threads can be accessing it and they are race conditions and bad things can happen and we know what kind of bad things can happen.

Similarly, this cache itself is shared and so, you know for example, that it can happen that one thread comes in it says is the requested file not in cache. Both of them figure out that it is not in cache and they have duplicate entries of the same file. Worst still it can happen that you know some schedules of this code can cause empty entries in the cache.

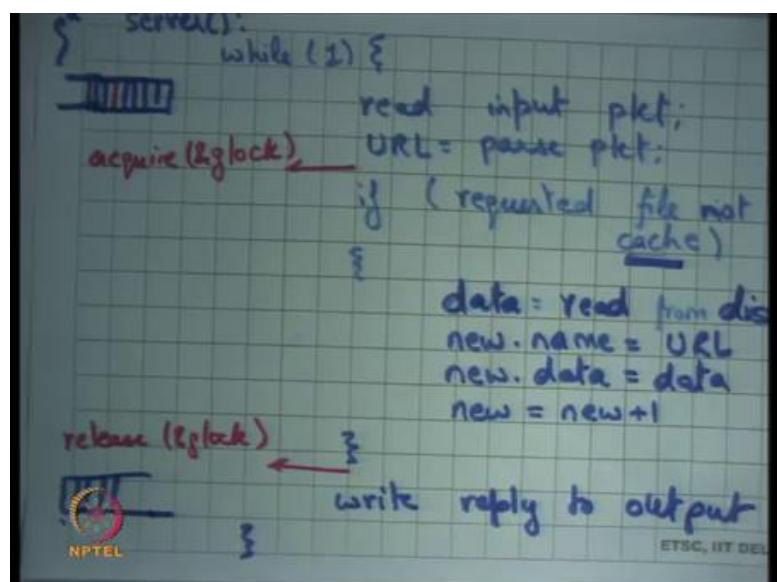
Or some it can even happen worst that it can so happen that you have the URL of one page, and you have the content of another page right all these possibilities right. So, what

do you want? Do you want to use some kind of locks right, you want mutual exclusion basically right and the way we know about how to solve mutual exclusion is locks?

So, what I am going to do is I am going to say let us say; let us say I do and acquire. Let us say I have global lock let us say this lock basically maintains a mutual exclusion for the whole cache and then I have a release lock right. This will take care of concurrent access to the cache; this will not take care of concurrent accesses to the input and output queues.

Let us ignore the input and output queues for some time let us say they are special you know special data structures and we do not talk about how these input and output queues are implemented. But, let us just talk about first the shared concurrent access from the cache. So, you acquire the global lock and you release the global lock is this an acceptable solution have we made any progress from our previous case, so I see some heads nodding, so why not.

(Refer Slide Time: 25:52)



So, what is happening, basically let us say now you have multiple threads ok, multiple threads could be servicing multiple requests simultaneously right. So, each thread is servicing one request, but if there is even one cache miss then you are holding the lock and you are accessing the disk. And so, for that period of time nobody else can take the lock, so what has happened is instead of the request having in the previous case the request was waiting at the input queue.

Now, the request is waiting at the lock acquisition that is the only difference the waiting is still the same. I still once again the problem is if I am standing behind a cache miss, then I will have to wait because the cache miss has taken the lock first and he is going to take a long time. And even though my request would have been would have taken less time. We are basically serialized completely serialized, so this the global lock will definitely not solve the problem.

In fact, you know we have just made things more complex without increasing any performance at all. So, here is an example we would definitely need fine-grained locking to have any benefit of concurrency alright. And so, what kind of fine-grained locking we are we going to have well, what are the shared resources you have these cache pointers new which are pages in the cache.

You have the disk itself is the shared resource right, so when you access the disk. The disk should not be accessed concurrently simultaneously you know accessing with the disk it requires sending some commands to the disk using. Let us say in and out instructions and so, those in and out instructions need to be mutually exclusive.

So, the disk is exclusive the each of these cache entries need to be exclusive, and this check needs to be you know atomic with respect to the other operations here right. And so, you know what the exact solution is basically you know is basically this is a very common scenario where you are maintaining a cache and you need to maintain consistency etcetera.

And we are going to look at this in very a lot of detail when we study file systems. So, file systems have a similar problem you know when you basically accessing your files not each and every request does not go to your disk. It is actually cached in the memory and you have to synchronize between multiple threads making multiple access concurrent accesses to multiple files. And you know here is a sketch of a solution you can basically say I will have a lock for the whole cache here, but then I will try to release a lock.

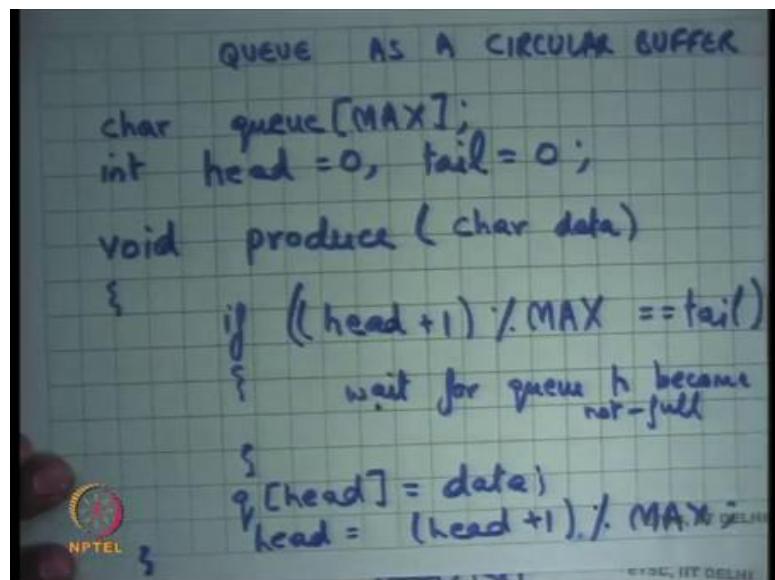
I will only use a lock as soon as I will release a lock as soon as I have done this check, but before I release this lock, I will take I will also have a per entry lock. So, I will have a full cache lock and then I will have a per entry lock, and I will you know after I have done the check. And before I release the global lock, I will take the local lock local lock

is a per entry lock and so, this area can execute concurrently for multiple threads and, so you have more concurrency alright.

So, and then you know you have to worry about other things also. So, you have to basically do fine grained locking and there are some ways you can do it and I just sketched one of the ways you can do it and we are going to see this in more detail. And so, you know just to motivate that fine-grained locking issues are actually a little bit complex all right ok. So, now, let us look at let us look at the queues right.

So here is the queue and here is another queue. And once again having queue is a very common pattern in systems in general and in operating systems in particular. And how are queues implemented well you could implement a queue circular buffer right.

(Refer Slide Time: 29:07)



So, a common way of implementing a queue is a [cir/circular] circular buffer right, so let us let us write some code to basically implement a queue. So, firstly, why do I need a queue? I need a queue because there is a network thread that is adding to the queue. And there is a server thread or there are multiple threads which are removing from the queue here. Similarly, there are there is a server thread that is adding to this queue and a network thread that is removing from this queue.

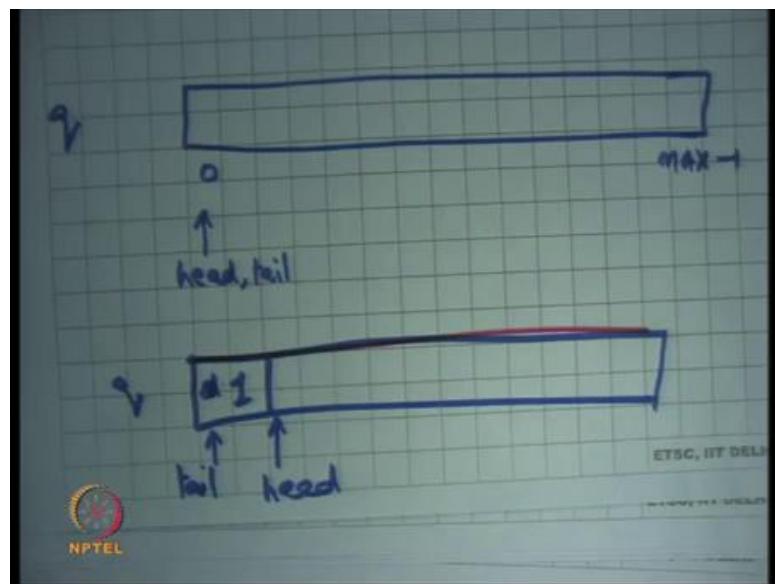
So, the logic is basically, and it is a first in first out queue right a queue basically by definition means it is the FIFO, the First in First Out. So, one thread can add to the queue

and another thread get remove from the queue and you will get a FIFO ordering on that. And let us see how a queue is implemented as a circular buffer I know I am sure you all of you have seen this before.

But I am just going to write the code, so that you know when I write about it in the concurrent code you have more context. So, let us say I have an array of characters which is maintaining my queue, and the maximum size of this queue is max alright. And then I have two pointers inside this array int head initialized to 0, let us say and a tail also initialized to 0 right, all these three variables are global variables alright.

Then you have a function called produce depending on who is the producer you know he is going to call produce input queue network is the producer output queue serve as the producer. In either case you basically say produce let us say some character which is your data and you going to say if alright.

(Refer Slide Time: 31:04)

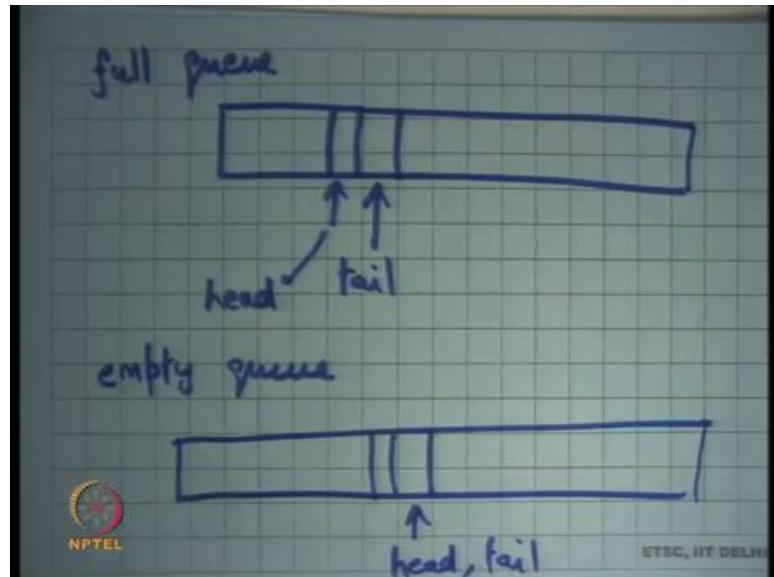


So, let me just first draw the queue just to make sure we understand this. So, let us say this is my array, this is queue right, and this is element 0 and this is element max minus 1 right, initially this is head, and this is also tail alright.

Now what will happen is that when I produce something I am going to increment head so, producing, so when I produce an element in the queue head is going to point head is going to advance and tail is going to be here and this will be the first element right.

Similarly, producer can keep producing and the head will keep incrementing till head becomes equal to tail minus one right in a circular way. So, I just keep incrementing head and, so the consumer may also increment tail and the queue becomes full.

(Refer Slide Time: 32:19)



So, the condition for full a full queue looks something like this tail is here right and head is just behind the tail right. This is a full queue, if I try to add one more data item I cannot do it without having to overwrite something that has not yet been consumed.

So, this is a full queue right, what is an empty queue? What is an empty queue? Tail and head are equal right. So, if $\text{head} = \text{tail}$ then that is an empty queue, basically the tail has basically hit the head. And now, I cannot move any forward for anything you know I cannot consume anything there is nothing forward ahead that is useful alright. So, what will produce do it will check if the queue is full if it is full then it will let us say return an error.

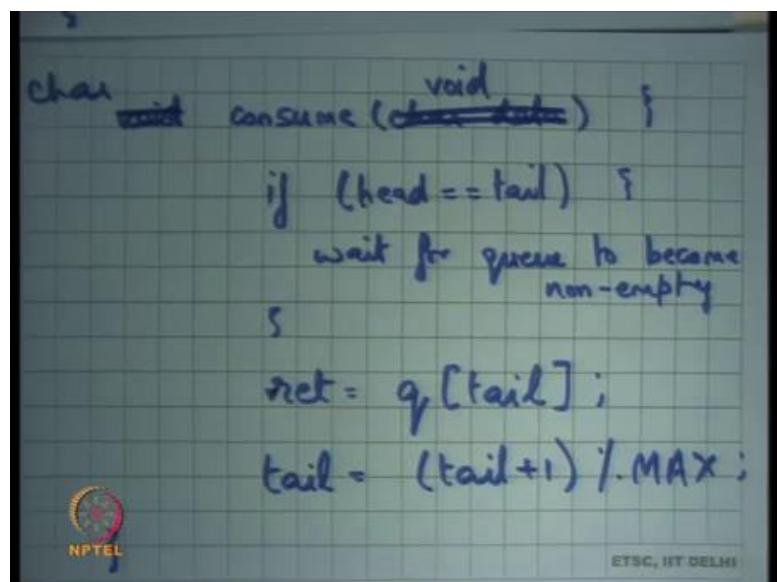
So, it will basically say if $\text{head} + 1$ and I will do it in a circular way modulo max is equal to tail. Then now let us say error here right let me just leave that as a blank otherwise I just say $q[\text{head}] = \text{data}$ and $\text{head} = (\text{head} + 1) \% \text{max}$ right.

And similarly, you can write the code for consume which I am not going to write here right. So, what should you do here? If you are just operating as a single thread and you

call produce, then you may want to say I want to give an error here right. So, you may say error here you may say error at this point.

But if you are executing in a multiple thread world and you have a shared queue which is shared by multiple people you may want to say let us wait right because there are other threads that are running at the same time. And so, it is likely that even though the queue is full right now it will become empty in some time right. So, one option is wait for queue to become not full right ok.

(Refer Slide Time: 35:00)



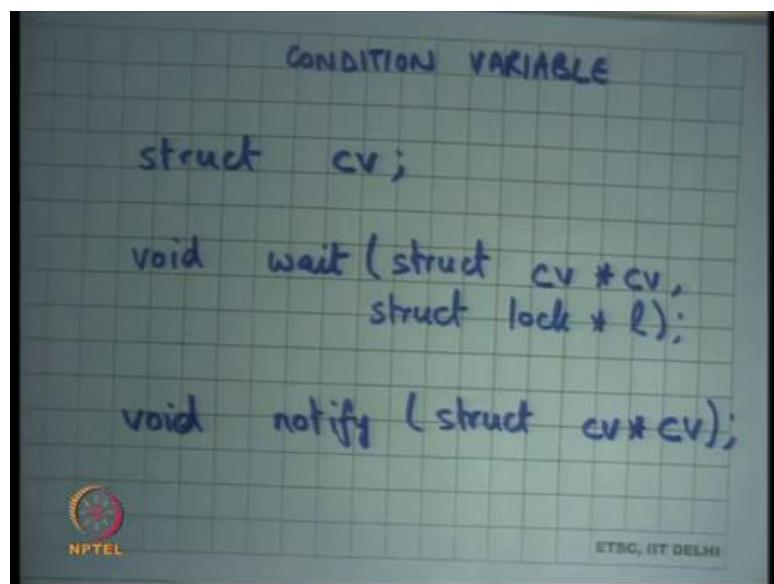
Similarly, in the consume I could have a similar thing here, if head is equal to tail then wait for queue to become nonempty alright. And so, sorry the consume has a void argument and returns a character right you can say return value is equal to $q[tail]$ right. And $tail = (tail + 1) \% \text{max}$ alright ok.

So, now my question is how does, so there are multiple threads one thread is calling consume one thread is calling produce it is possible that the produce finds the queue full and it waits for the queue to become not full. Similarly, it is possible for the consume to become and so, for the queue to become empty and for the for a thread to wait for it to become nonempty. This kind of synchronization this is also solved for synchronization.

So, I am waiting for a condition to become true and only when it becomes true will I be able to move forward right. And the constructs that we have studied on that abstraction

that we have studied so far of locks is not enough to implement this synchronization alright. It is not possible with locks to say that you know I want to wait on a condition locks are only for mutual exclusion; I cannot use a lock to say I want to wait on this condition to happen right ok.

(Refer Slide Time: 37:10)



So, what am I going to do? I am going to define a new abstraction that is called condition variable alright. So, let us understand what this abstraction is just like a lock there is a type called struct CV let us say just like there were struct lock and you can instantiate a lock with this type. Similarly, you have a type called CV and you could have you could instantiate this type and then you have two functions void, wait alright. struct CV star let us say CV and then and there is another function called notify struct CV star all right.

Actually, the wait has two arguments and the second argument I am not writing right. Now, Before I explained why we need the second argument, but let us just understand the spirit of this abstraction.

(Refer Slide Time: 38:21)

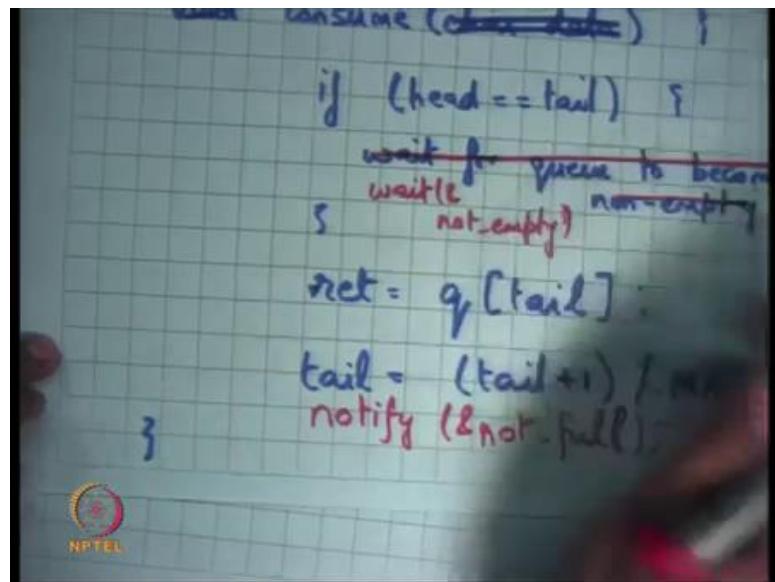
The image shows handwritten C code for a producer thread. The code defines an integer `head` and `tail`, and a structure `CV` with members `not_full` and `not_empty`. A function `produce` takes a character `data` as input. Inside the function, there is a check for overflow (`((head + 1) % MAX == tail)`). If true, it adds a note: "wait for queue to become not full". It then performs a `wait(¬_full);` operation. After the wait, it sets `q[head] = data`, increments `head` (`head = (head + 1) % MAX;`), and performs a `notify(¬_empty);` operation. The code ends with a `void notify()` declaration and a `consume` function declaration.

```
int head = 0, tail = 0;
struct CV not_full, not_empty;
void produce (char data)
{
    if ((head + 1) % MAX == tail)
    {
        wait for queue to become not full
        wait (&not_full);
    }
    q[head] = data;
    head = (head + 1) % MAX;
    notify (&not_empty);
}
void notify () struct
{
    void
    consume (char data) {
}
```

So, the spirit of this abstraction is in the previous example the programmer can define two condition variables and the produce let us say. So, I could define a condition variable called, so that is a struct CV there are two condition variables not full and not empty right. And I could say, so this English sentence wait for queue to become not full.

I can now replace with something; that means, similar thing let us say wait on not full on this condition variable called not full right. So, I have defined a condition variable and basically said now let us say I am going to wait on this condition variable called not full right. Now, how am I going to know whether it has become not full the consumer has to tell me right.

(Refer Slide Time: 39:19)



So, the consumer will basically here say notify not full ok. So, that is how you connect the two basically say that if you find the queue to be full you basically wait on this condition variable you know that I have called that I named not full. And then when a consumer consumes something then clearly after it has consumed something the queue is not full. So, he can say notify not full and the idea is that when he says notify is going to wake up any thread that is waiting on that condition variable right.

So, those the semantics are then when he calls notify if at this point there is any thread that is waiting on this condition variable called not full and get very woken up and it be able to proceed alright. Similarly, you know symmetrically instead of wait for queue to become not empty I could say wait and not empty right and here I could say notify not empty ok.

The effect of notify is that if there is a thread that is waiting on that particular if there a thread that is waiting on that particular condition variable, then it will get woken up it will return from wait. On the other hand, if there is nobody who is waiting then it will have no effect at all right, it is just a no right.

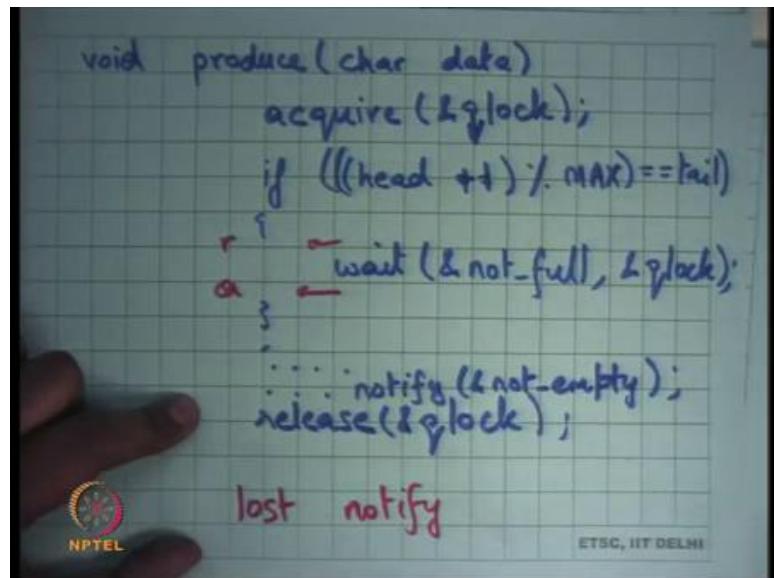
So, it is possible that actually at this point there was no other thread that is waiting on the not empty. In fact, the queue was not empty at this point we are just calling notify unnecessarily it was actually the queue had let us say ten elements and the both the producer and the consumer happily going along. But you are just calling notify

unnecessarily here right, but that is it is not incorrect. It is not it may not be efficient, but it is not necessarily incorrect right, so there are some problems with this code.

Firstly, these are shared variables right q is a shared variable, head is a shared variable, tail is a shared variable. And the two at least two threads one is a producer and one is a consumer and both of them are accessing these shared variables. And they are not doing any sort of mutual exclusion in their accesses and, so bad things can happen right ok.

So firstly, I need I also need locks apart from condition variables I also need locks in this particular code right, because these shared variables that are being accessed. So, not only do I need this kind of wait notify synchronization I also need locks to be able to maintain mutual exclusion on my accesses to these shared variables right ok.

(Refer Slide Time: 42:06)



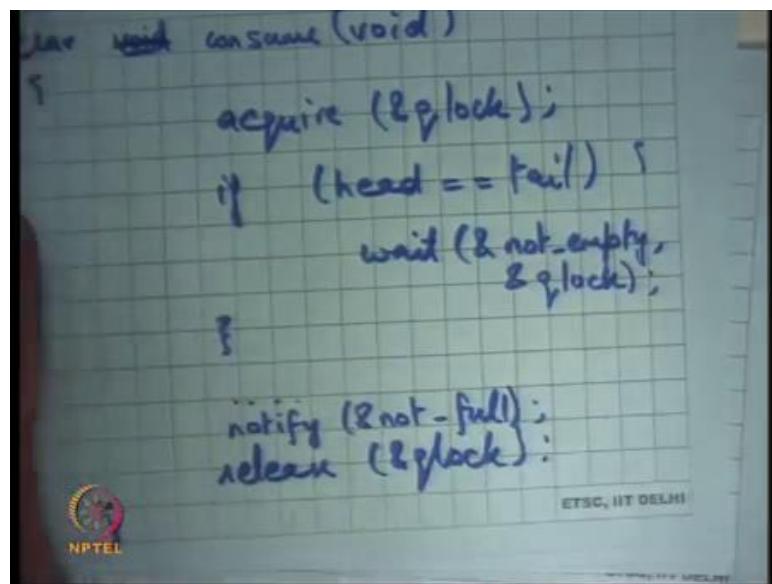
So, let me rewrite this someone say void produce char data and once they acquire. Let us say I have a lock called qlock right, and then I say if head is equal to tail sorry head plus 1 mod max is equal to tail then wait on not full. And I am going to leave it empty because I am going to say something else later. And then I am going to say you know whatever was earlier cause to produce something and then I return.

So, that is my produce code alright and of course, I release the lock here right. Acquire the lock I do some operations if the operations require me to wait, I wait and then I get notified and then I release a lock yes.

Student: Sir, while waiting do we need to just release the lock because we were switching to.

So, the question is while waiting do I need to release the lock yes, so I am going to come to that very soon alright, so good.

(Refer Slide Time: 43:43)



So, let us say similarly void consumed is are let us say char consume a void is basically going to say acquire. It is also going to acquire a q lock it is going to check if head is equal to tail then wait on not empty alright. And then do whatever you like and then release q lock right.

So, now, as was pointed out there is a problem with this code one thread acquires the q lock starts waiting finds the queue full starts waiting on this condition variable called queue not full, but it has not released a lock right. So, if it has not released the lock there is no hope that a consumer will be able to get the lock.

And so, the consumer will get start waiting here, the producer will start waiting here, and you have a deadlock right. So, what did you want to do? You wanted to release the lock before going to wait alright. So, one option is to call release here right, so let us say release now say r here and then do I need to re acquire the lock when I come out of a wait?

Student: Yes.

Yes, because you know I am going to have to execute this, so I need to reacquire, so I call release before wait and call acquire after wait.

Student: Can we make them atomic.

Alright, so you know, so there is a suggestion can we make them atomic. So, the so firstly, this is this sounds this sounds I release the lock then I go to wait then I come out of wait somebody called me notify I come out the first thing I try to do is acquire the lock. And then I get the lock then I can you know I can just go forward.

There is a problem as there was very rightly pointed out that these operations of releasing the lock and actually going to wait are not atomic with respect to each other; let us see what something bad that can happen. Let us say the producer basically comes in finds a queue to be full he checks; his condition finds a queue to be full he releases a lock he releases a lock.

And he is just about to go to wait before he goes to wait consumer comes, in consumes the element signals not full, but that signal will be wasted right. So, that signal will be wasted, and so, the consumer has sent it signal on that notify. So, it will basically say notify that notification will get wasted and then the producer will go to sleep.

So, it is like saying that I have released the lock, consumers can now come in consume the entire queue you know and send signals. But you know I because I have not yet gone to sleep those signals are those notifies do not mean anything to me, then I go to sleep right. And now, I will keep waiting forever the consumers have gone done their job they are far as far as they are concerned, they have notified. The producer was you know, has not started going has not started sleeping at that point and so, you know he is basically.

So, you know one analogy to this is that let us say you know you basically release the lock which means you allowed other people to come into your house, but now you have to go to sleep. So, you other people have come into their house and they have you know they have changed some things and they have also said you know wake up this also triggered the wake-up alarm.

But you know you have not really reached your bedroom and so, you have not started sleeping after everybody has gone then you go to sleep and you will keep sleeping

forever basically you know. So, that is the and so, let us say this is the lost notified problem right that is a bad example, but.

Student: [laughter].

At least I hope it is as this trying to give a physical counter analogy to this thing alright. So, this is the lost notify problem right, so basically the notification is getting lost alright. So, the abstraction is basically the condition variable basically takes two arguments a condition variable and a lock right. And the semantics are that weight goes to sleep on the condition variable and releases the atomic the lock and does this in an atomic fashion right.

So, basically it does all these three things it releases the lock and goes to wait right. So, inside inside this I will basically give this the second argument queue lock. And the semantics are that it will release the queue lock and it will go to wait and it will make sure that nobody can call notify in the middle of these two things right.

It is not possible that when after I release the lock and before I actually went to sleep nobody can call notify, so it is atomic in that sense right. So, you basically going to release the lock and wait, and it is going to be atomic nobody can call notify in the middle of those two operations.

Similarly, the other abstraction is that as soon as it gets notified the first thing is going to try to do is to do a lock acquire right. Lock acquired need not be atomic I mean it is just you know you just going to try or to acquire the lock, and you may have to wait for being able to acquire the lock right.

In fact, this code was not complete, so before you release you basically also say notify not empty alright and here you say notify not full, and here you have to release the lock. So, what will happen is the let us say a producer figures out that the queue is full it goes to wait on not full, but it also releases that lock atomically.

So, if somebody is able to get the lock by that time the producer would have slept definitely right. So, only after it has gone to sleep will the consumer and has released the lock will the consumer be able to acquire the lock it will do all its operations, and it will signal not full.

When it signals not full it basically looks at all the threads that are waiting on the not full in this case there is only one thread that is waiting on not full and it is going to wake it up. The threads going to get woken up and the first thing it is going to do is going to try to acquire the lock. Is it going to get the lock immediately?

No because you know in this code here the release you know the producer has the consumer has not released the lock yet. So, even though I have notified the producer the producer has just woken up and the next thing is going to try to do is to acquire the lock.

In this code it is possible that this producer the consumer producer gets to run before the consumer actually gets to execute the next statement. And so, it is possible that the log is still held by the consumer, so he is going to try to acquire the lock he will not get immediately. But eventually he will get it because this one is going to release it, and so now, he can get it and he can continue it is operation right.

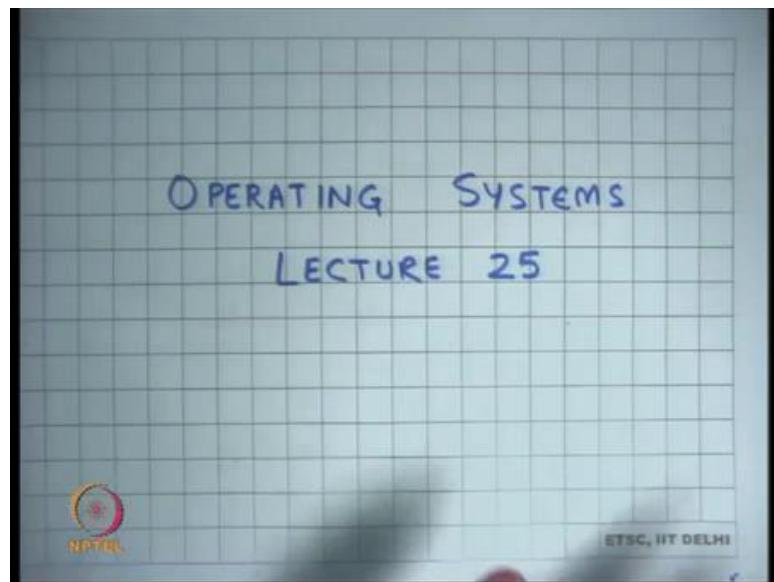
So, the semantics of wait and notify that wait takes two arguments the first is a condition variable let me just put this on. So, the first is the condition variable and the second is the lock. Semantics are that the thread which calls wait on these two arguments will go to sleep on CV and release the lock L in an atomic fashion right.

The semantics of notify that it will wake up all the threads that are sleeping on CV alright and that is it, if there are no threads waiting on us sleeping on CV notify will I have no effect right. The other semantics of waiter that as soon as you wake up from sleep then the first thing you will try to do is reacquire L right, so alright. So, let us stop here and continue this discussion next time.

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

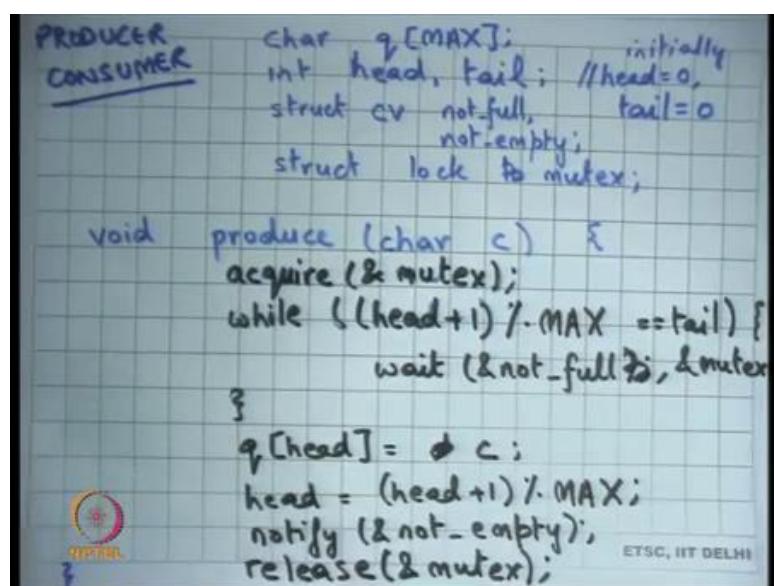
Lecture – 25
Multiple producers, multiple consumer queue; semaphores; monitors

(Refer Slide Time: 00:25)



Welcome to Operating Systems lecture 25, right.

(Refer Slide Time: 00:30)



So, last time, we were discussing a Producer Consumer example and we were you know the example that we had was there is a network thread and there is a server thread and there is an incoming queue of network packets and there could be multiple server threads for example, and similarly there is an outgoing queue and then, there could be multiple server threads and there is a there is a network threads that is picking packets from the outgoing network queue and putting them on the wire.

In any case this kind of a pattern where there is a producer and there is a consumer, so there are some producer threads and there is some consumer threads and there is a shared queue in the middle and the packets are being processed in the q in a 5 FIFO order; first in first out and the producer is supposed to produce elements and the consumer is supposed to consumer elements from right.

And we said that there are synchronization problems with that; of course, because number 1, the q itself is the shared structure. So, all accesses to the q and to the pointers in the q, have to be protected using locks because you know shared accesses to shared, I mean concurrent access has to shared data is dangerous. So, you should protect it with a lock. Not just that, you have to make sure that a producer does not produce to a full q and a consumer does not consume from an empty q right.

So, those are the two conditions that you have to satisfy. And so, not just not only do you have to maintain mutual exclusion, you have to also maintain these two invariants that you know that the producer should not produce to a full q and a consumer should not consume from an empty q, all right.

So, let me take go through this example in a little more detail. So, let us say this is my; this is my circular array that I am using to represent a q. It has MAX elements. I have 2 pointers: head and tail. Initially, I set head and tail to 0 and then, you know we looked at it last time and we said to ensure that a producer does not write to a full q, the producer should check for the condition and if the condition is not true, then it should wait on this condition variable called not full right.

And somebody will basically signal or notify this condition variables and so then, the producer can ensure you know then proceed. And similarly, the consumer should check if the q is empty and if it is empty, then it should wait on this condition variable called not-empty right and then, there should be a lock which basically does mutual exclusion

and today, I am going to I am calling this lock, I mean the name of this lock variable is mutex.

You could name it anything, where it is very common to name it mutex, to make it clear what the meaning of this lock is. This means this lock is basically for mutual exclusion and it is common; it is common practice to use the word mutex to represent a lock which is basically being used for mutual exclusion all right ok.

So, let us see how our producer works. The producer needs to first acquire the lock because it is going to operate on the read and write share data. So, it acquires a mutex and then, it checks whether the q is full and, in our case, you know because it is a circular buffer checking if the q is full involves checking if head plus 1 board MAX is 0 tail.

And notice because I have the lock, you know I can do this you know without having to worry about concurrent accesses, concurrent changes or reads or writes to head at all right. And if not, then I should wait on this condition variable called not full and the second argument to the to this wait is basically the mutex and the semantics of the wait are basically that, it will release the lock or release the mutex just before going to sleep in an atomic fashion and after it gets woken up, it the first thing it will do is try to reacquire the lock right.

So, basically the idea is that it will release the lock before sleeping and it will reacquire the lock, after it wakes up right and the condition that is waiting on or the condition variable, it is waiting on really is this is called not full right. Now, if it gets woken up by somebody. So, you know somebody calls notify or not full, what will happen?

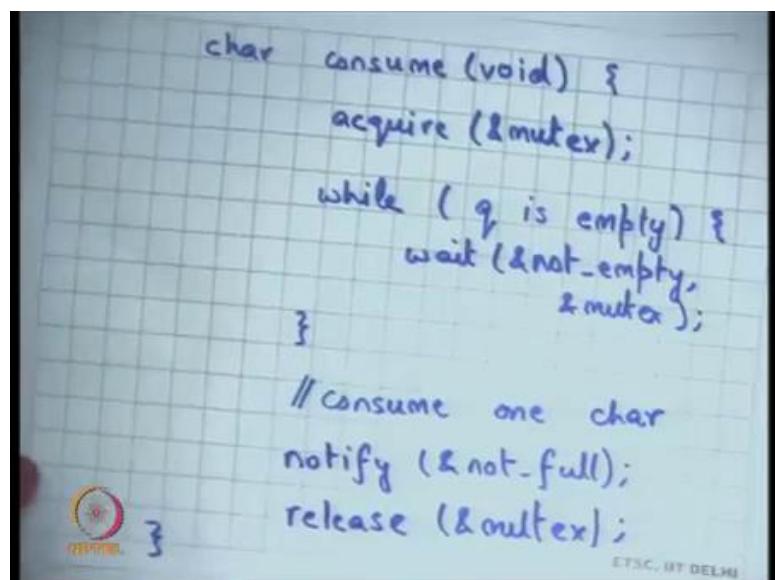
It will eventually, it will the next thing we will tried to do is try to reacquire the mutex and if it is able to reacquire the mutex, it will come out of the wait. In this code, the first thing you will do is you will check the condition again right. Why do I need to check the condition again? Let us hold that discussion for a moment.

So, let us say you know you check the condition again and you know assuming there was just 1 producer and 1 consumer and the consumer has just notified me, basically means that there must be some empty space in the q right; somebody has just consumed something and he has notified me.

So, there must be some empty space and so, this condition should be false right and so, you should get out of the loop and at this point, you can be sure that there is empty space in the q and head is pointing to that first empty slot and you produce in that empty slot right. After you produced in the empty slot, you may want to notify the consumer. In this case, I am actually notifying every time.

But you know you may want to be more efficient; you may want to say if you know if q has 1 element, only then notify right. Do not keep notifying unnecessarily, only if you are if you have just transitioned from 0 to 1 now or you are transitioning from an empty q to a non-empty q do you need to notify the consumer and consumer will have a very symmetric code right.

(Refer Slide Time: 05:52)



So, if I just look at the consumer code, you know same thing you acquire the mutex, you check if the q is empty. This time I am not writing it in English instead of the actual condition right. I will wait on not-empty and mutex just like before at this if I come out of wait, I will recheck the condition and if I add just 1 producer and 1 consumer, this condition will always be false.

And so, I will come out of the loop and I have consumed 1 character, notify not-full release the mutex all right. So, now why do I need this loop? Right. Can I be sure that if the producer comes out of the wait? The q is definitely not-full all right. So, somebody, so, you said no, why?

Student: Because there are multiple producers.

So, yes. I mean if there are multiple producers, it is possible that even if you come out of the wait, the q is still full right. Why can it, how can it happen? Let us say you know there were 2 producers, both of them found the q to be full and both of them start waiting.

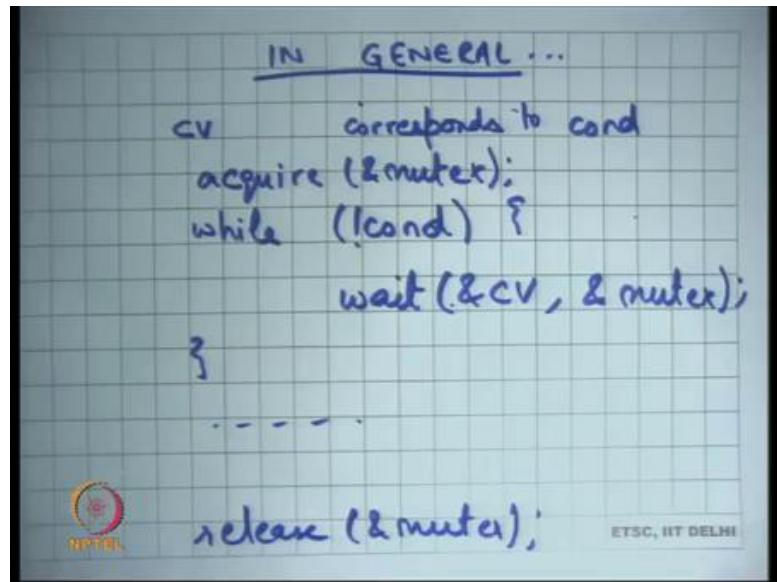
Now, one consumer comes along, and he consumes an element and then, he calls notify will have the effect of waking up both the producers let us say right. And so, both the producers will wake up; one of the producers will be able to get the mutex, the other producer will wait on the mutex. The producer who gets the mutex will you know check the condition; he will find it false; he can produce 1 element.

But at this point, he has made the q full. When he releases the mutex, the second producer will be able to get the mutex. He will come out of it and it is not ok for him to just go and start producing, it is he should again check the condition. So, the invariant is not that when you come out of it where the condition has definitely become false right. You went to wait because the condition was true, but it is not necessary that when you come out of wait, the condition is necessarily false.

So, hence you need to recheck the condition after you come out of wait right and so, you need a while loop instead of an if which we had last time. On the other hand, if you had only 1 producer and 1 consumer, it would have sufficed right, but it is always safer to you know use while I mean just to make sure, I mean then you are basically relying on more invariants in your code etcetera right.

So, it would have sufficed if you had just one. So, instead if you had to adjust 1 producer and 1 consumer, you could have replaced this if while within if ok, but if you have multiple producers, then you need a while. Similarly, if you have multiple consumers, then you need a while here right all right. So, this is a very common pattern. So, let us understand how a condition variable used in general right.

(Refer Slide Time: 08:39)



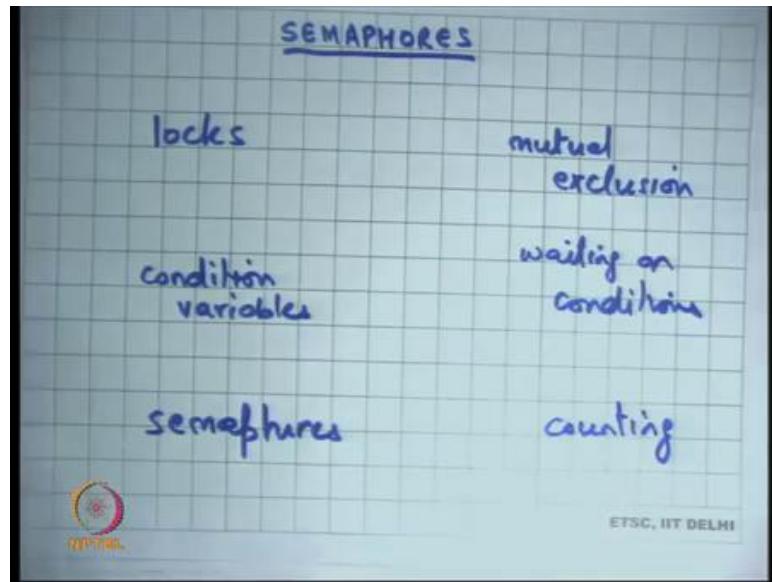
So, in general. So, I have taken an example of a producer consumer example to illustrate how condition variables are used, but in general a condition variable cv corresponds to some condition; let us say cond right. And you would typically want to check the condition, and you would typically want to check the condition in some kind of a loop.

So, while not cond, you will wait on cv all right. And because you know you want to make sure that the checking of the condition and the act of actually waiting should be atomic with respect to each other and also you want to make sure that there is mutual exclusion maintained in your checking of this condition.

You would typically want to also use a lock or a mutex right and that is the mutex you will pass as the second argument on this right. So, this will be the general pattern and of course, you will do something here and then, you will release the mutex right. And you may want to notify or not notify depending on whether you expect somebody else to be waiting on a condition or whether you are made some other condition true or not right.

So, this is a general pattern in which a condition variable is used. You take a mutex, you check a condition and if the condition is false, then you wait. But you do not, but you generally put the wait inside a while loop so that when you come out of it you recheck the condition right. Because the condition I mean may have become false by the time you actually came out of wait, just like in the producer consumer example right, so general pattern.

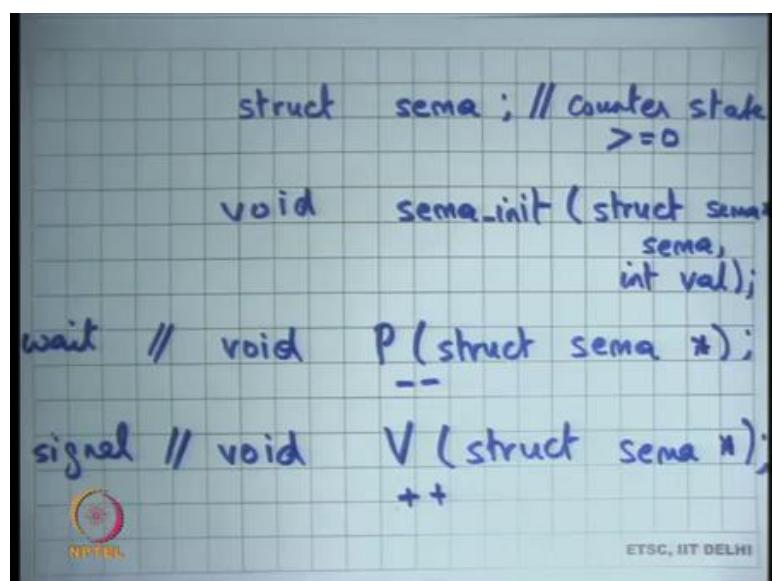
(Refer Slide Time: 10:30)



So, now I am going to talk about another abstraction which is Semaphores. So far, we have seen two abstractions to do synchronization; one is locks and locks were basically used for mutual exclusion, another was conditions, condition variables. These were basically associated with conditions. So, waiting on conditions all right.

Semaphores is yet another abstraction and they are basically you know the abstraction if you know more most concisely described as such something that enables you to count right and I am going to discuss exactly what semaphores mean right. So, we have looked at these two abstractions already. And today, I am discussing semaphores and let us see what semaphores are?

(Refer Slide Time: 11:30)



So, semaphores you know basically a common pattern in programming is basically like this programmer, producer consumer example involves counting of resources right. So, in the producer consumer example the resources were the slots in the q right. So, there were let us say MAX slots in the q and you make wanted to make sure that you know the you never go to MAX + 1. So, you never know; so, those are the number of sources and you are trying to count the number of resources.

So, the producer has to count the number of full slots and similarly, the consumer has to count the number of empty slots and so on. And so, this is a very common pattern and in 1965 Dijkstra, whom you probably also know from a shortest path algorithm, you know proposed this subtraction called semaphores right. So, semaphores are basically in order to defined by a type, let us say it is a struct sema right.

Just like there were struct lock and struct cv, there is a type called struct sema and this is a state full type all right. Let us see and then there are 3 functions; there is void sema init that sema and int val. So, sema init takes the first argument as the semaphore variable sema and the second argument has an integer value which must be and basically it sets the counter inside sema to be equal to value.

So, it just initializes the semaphore to that counter called value right. So, basically semaphore has a state called counter right and the initial init variable will initialize the counter to value to val; assuming values greater than equal to 0. So, this counter should be greater than equal to 0 that is another invariant of a semaphore all right. Then, there is a function called P and there is a function called V.

The alphabets P and V are you know are the first alphabets of the meanings of these functions in Dutch, you know which is basically Dijkstra's mother tongue, I guess. But you know the other names of this are wait sema wait or sema signal right, but we want to just call them P and V right. So, let us see what P and V mean? All right.

So, P's semantics are that it will decrement the counter of the semaphore by 1 right and V semantics are that it will increment the counter of the semaphore by 1 right. So, this is a minus-minus and operation and this is a plus-plus operation basically, all right. Except that minus-minus will only happen if the counter was greater than 0; if it was equal to 0, you will not you know so you will never allow the semaphore to become negative.

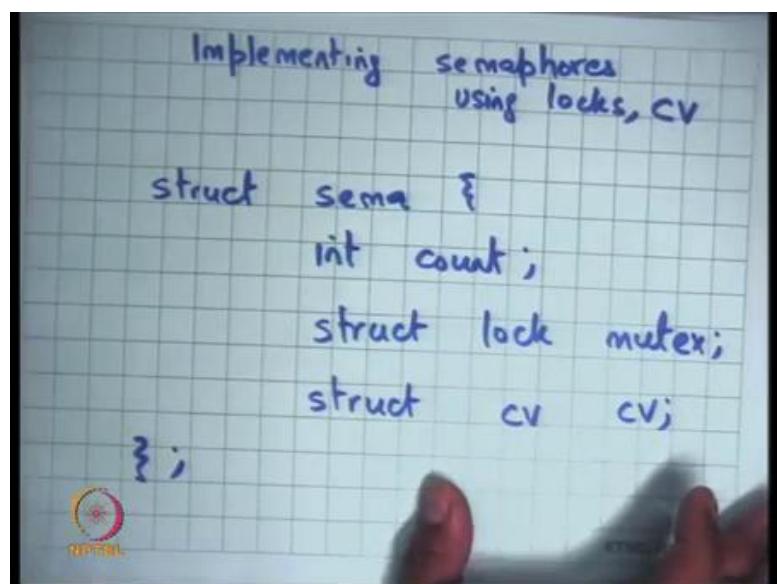
So, if it is equal to 0 and you call P on semaphore that is equal to 0, then you will wait and you will wait for it to become greater than 0 and only when it becomes greater than 0 value, do the decrement operation all right ok. Once again, init just initializes the counter to val. P decrements the counter except that it ensures that the semaphore never becomes negative.

If the semaphore is 0 and you know if somebody has called P that which means it may become negative, it will not actually decrement it, it will start waiting and we will start waiting for the semaphore to become greater than 0 and when it becomes greater than 0, at that point, it will decrement it and come out of P right.

And V is simply going to increment the semaphore. And of course, you know when it increments the semaphore, if it finds that somebody is waiting for it to become greater than 0, then it will wake it up and say you know why do not you know you can now decrement it is that right. So, those are the semantics.

Most importantly these operations, all these 3 operations are atomic with respect to each other right. So, this operation of decrementing and waiting is atomic with respect to increment and initialization and all you know. So, all these are atomic with respect to each other right. So, that is why this is these are these an interesting abstraction from a concurrency standpoint. So, P V and init are basically completely atomic with respect to each other, all right.

(Refer Slide Time: 16:25)

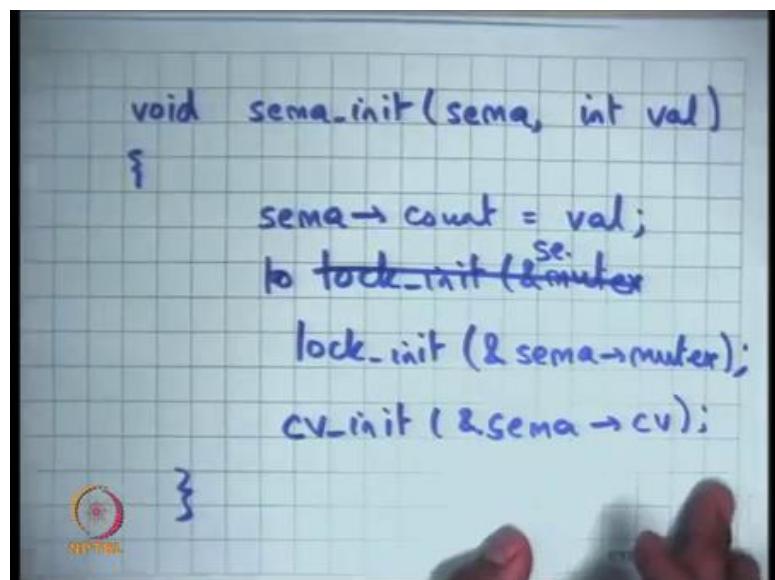


So, in order to make this clearer, I think it will help if we implement semaphores. you know the implementation of the semaphore will make it clear exactly what the semantics are the semaphores are in you know in concrete terms instead of just plain English and I am going to use locks and cvs to implement condition variables to implement semaphores right.

So, you know what I showed you in the previous slide was the semantics or the abstraction of a semaphores. Well, let us look at how a semaphore could be implemented, and I am going to give you one example implementation of a semaphore right. So, let us say struct sema will have a 1 field which is the count right. This is an integer field and let us say it has a an associated mutex lock right and an associated condition variable ok.

So, that is let us say that is how I define my semaphore and implement my semaphore, it has 3 fields account variable which is used to maintain this account and then, I have these two other fields that are basically there are to ensure atomicity and of my access with respect to each other and also in you know implementing semantics and along allowing waiting right on a condition.

(Refer Slide Time: 18:03)

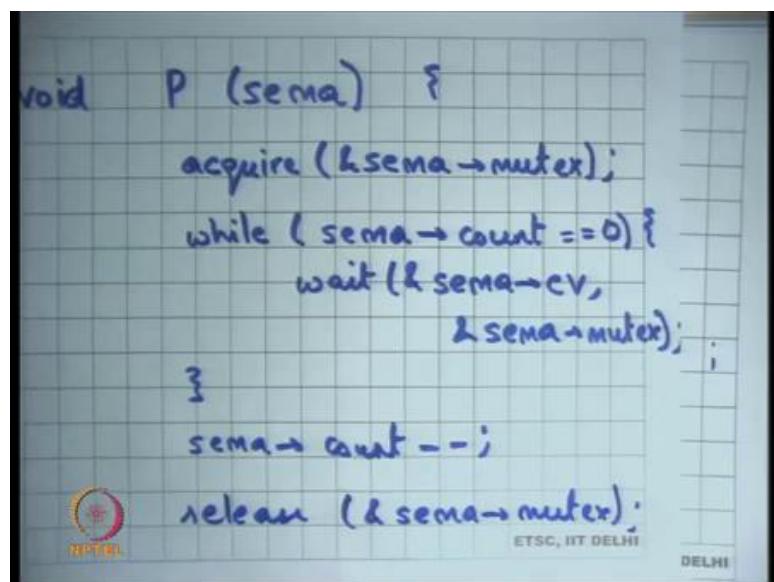


So, let us say I do this and then, I need to implement sema init. There are two arguments sema and int val and what I am going to do is say sema.count is equal to val. But before I

and also going to say lock or lock init. So, let us say I you know there is a function which initializes a lock and let us say and initializes it to you know unlock state.

So, it just sema dot mutex all right. And let us say there is another function called cv init, I mean just initializing the corresponding fields right. So, that is the that is an initialization function just initialize all these all these repeats; nothing great I mean I just initialize the count, and these are just initialization functions other corresponding components right.

(Refer Slide Time: 19:09)



Let us see P. So, I want to say P on sema, I could say I would say acquire. So, I want to make them atomic with respect to each other. So, I will say acquire sema.mutex right, that is ensured and nobody else could be operating on the semaphore at this time. So, all these operations should do acquire on sema.mutex and then, what should I check? I need to decrement the count right, but I also want to check that the count is greater than 0. So, what should I do? While; while what?

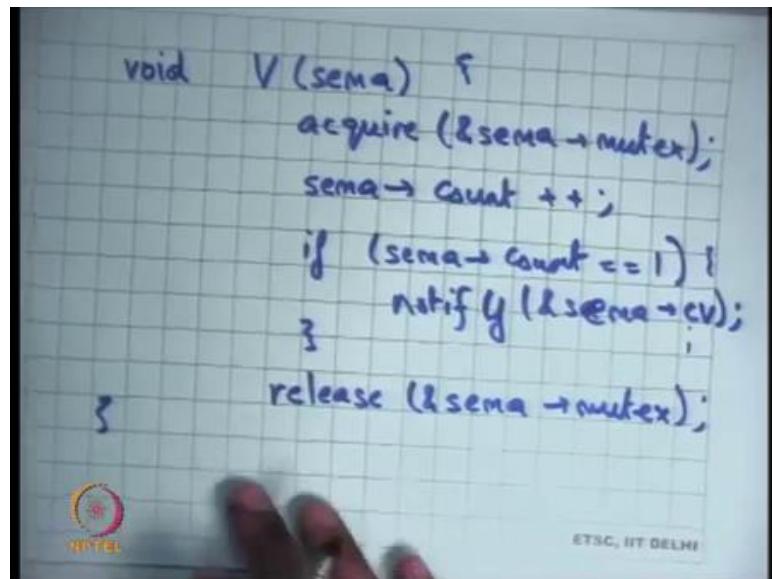
Student: Count is while count.

While count is equal to 0 right. So, I am using the condition variable to basically wait on the condition that discount becomes greater than 0 right. So, while count is not greater than 0 or while count is equal to 0 other in other words, I would say while (sema.count = 0) wait on sema.cv sema.mutex right ok.

And then, if I have if I come out of the loop, I can be sure that sema.count > 0 right and at this point, I can just say sema.count-- all right and then release.

Make sense, basically I wanted to do this, but I also wanted to do this only if this condition is false right and so, till this condition is true, I need to wait and I am using a condition variable to wait and I will and as you can imagine what I will do is in the V function, I will call notify on this condition wait right and of course, I use the mutex, I release the mutex before I go to wait and I do this in an atomic way just like all the other pattern we have seen so far, all right ok.

(Refer Slide Time: 21:36)



Now, let us write V.

Student: Sir.

Yes.

Student: Sir when we call (Refer Time: 21:40) pass the mutex is the argument.

Ok.

Student: So, the wave function that release the mutex on it is own, so then, why do we need to release it again? As in general, we happened on this but.

No. So, the wait function releases the mutex, but also reacquires it after waking up. It does not just release; it also reacquires it ok. So, at this point or any you know anytime you are outside wait, you would hold the mutex in this code right. Similarly, there is V on sema. What do I do here? First, I acquire the lock right.

So, I acquire sema.mutex and I need to do that to maintain mutual exclusion between P and V and between multiple V's and multiple you know. So, mutex is basically ensuring mutual exclusion between a P and a V or between multiple P's or between multiple V's basically right. So, it just making sure that things are completely mutually exclusive all right, and what do I write here?

Student: Increment.

I just increment right, I do not need to wait on any condition, I can just you know V the semantics of V as I have told you is just to increment the semaphores count. I just say sema.count++. Do I need to do anything else?

Student: Notify (Refer Time: 23:18).

Notify and one thing is I could always notify, or a more efficient thing could be to check if the count is equal to 1, only then notify right. If it had just become 1 from 0, only then there is a likelihood that anybody is waiting right. So, I basically say if sema.count = 1 notify sema.cv right ok. So, that so this implementation obeys the semantics of semaphores and I am using lots and condition variables to implement semaphores.

But of course, you know you could, you know you could, you may not, you may just want to implement semaphores using assembly instructions, like you may want to implement semaphores using the atomic exchange instruction, that we have seen right. That may be a more efficient way of implementing semaphores and I would like you to think about it at home right. How will you implement semaphores using just assembly, instead of using these high-level abstractions of locks and condition variables?

Definitely possible, after all these locks themselves are implemented using assembly instructions right and so, our condition variables. So, let us look at abstraction once again. So, we have a type called sema. It is a stateful type, in the sense that it has a state

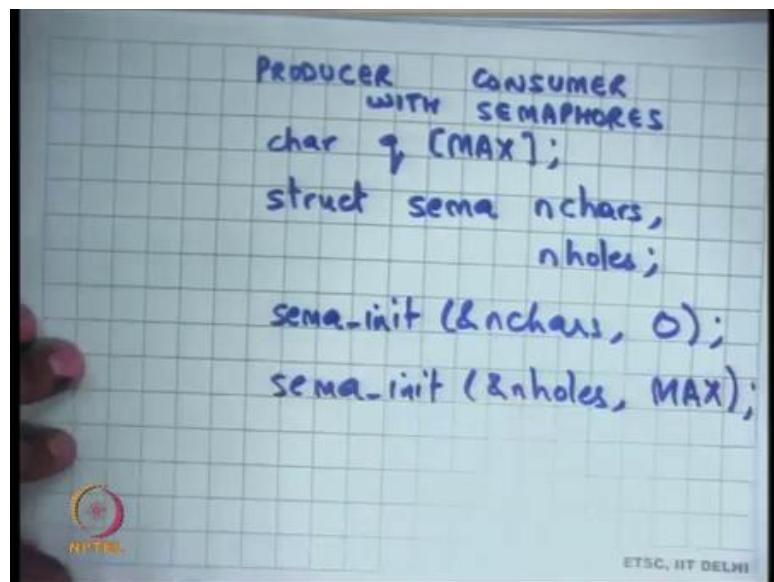
and the state is the counter right and the counter must be greater than equal to 0, non-negative.

And then, these are the 3 functions that you can use on the sema and this is a contrast to, so in many ways semaphores and condition variables look a little similar. Because condition variables also had a wait function and a notify function and one may be tempted to say or P looks very similar to wait and V looks very similar to notify right. After all, P is just saying let us wait on something and V is just saying let us signal something; except that we have now the difference between a semaphore and a condition variable as that semaphore also has state.

A condition variable did not have any state. If you call notify if somebody is waiting right now, he will wake up; if nobody is waiting right now, nothing will happen. If somebody goes to wait later, you know it says problem you somebody else you call him call to call notify from him. So, if a notify occurred before wait no they do not match up. On the other hand, with the semaphore if a V occurs before P, then it is because V were have incremented and so, P will not have to wait right.

So, in that sense, they do not there is because of the state they do not need to happen together, they can happen at different points of time. So, you know that this may become clearer, when we are even I talk about some example uses of a semaphores all right.

(Refer Slide Time: 26:15)

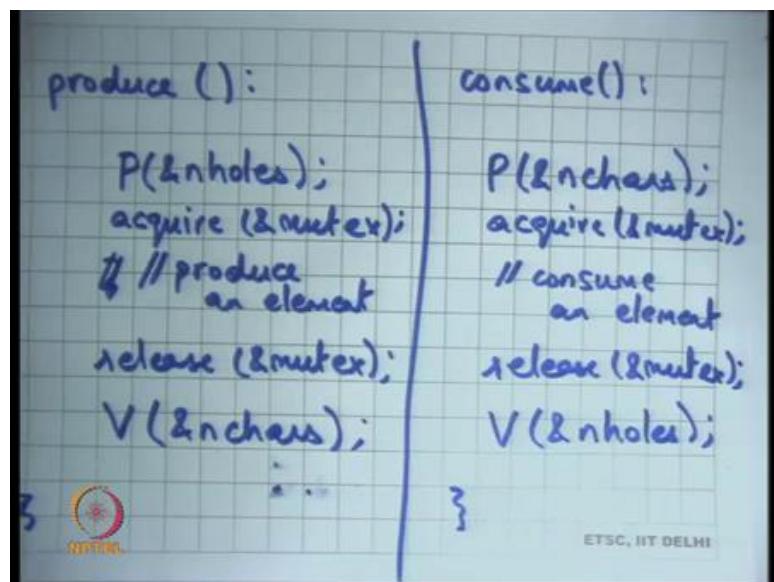


So, let us look at Producer Consumer with Semaphores. So, we have seen producer consumer with the locks and condition variables, but let us see producing consumers with semaphores and let us say we again have char q[MAX] and let us say I also have head and tail, but I am going to just you know abstract them and let us say now, I am as I basically I need to what I am going to do is I need to count the resources right, I need to make sure that a producer does not produce to a full q and a consumer does not consume from an empty q.

So, what I will do is I will maintain two counters; one for all for the number of elements that are present in the q and one for the number of elements that are or number of slots that are available in the q all right. So, I will have two counters and each counter will be of type sema; so struck sema, one is you know how many elements there are in the q? So, let us see n chars that is the how many elements there are in the q.

So, it is just a counter right structure, but I am using it semaphores to represent that counter and then, there is another counter called n holes which is how many empty slots there are in the q and the invariant will be typically that n holes plus n chars will be equal to MAX basically right. And so, I initialize n chars to 0, let us say the q was initially empty and I initialize n holes to MAX right.

(Refer Slide Time: 28:07)



Now, let us see how you write a producer. Let us say this is produce and let us say this is consume. All right ok. So, recall that in the previous case what I did was I acquired a

mutex, then I checked for a condition inside a while loop and then I waited and then, when I came out of the loop, then I consume etcetera.

But when I use semaphores, I do not have to do any of that; all I need to do is if I want to produce, I call P on n holes right. It basically means I am going I need you know decrement n holes by 1 an atomic manner, if I am and I because n holes are non-negative number, if I come out of P has to be non-negative number, when if I come out of P, I am sure that there is an empty slot available and the q right. And then, I produce an element.

So, let us say you know produce an element which may mean you know $q[\text{head}] = c$, $\text{head} = (\text{head} + 1) \% \text{MAX}$ and then, and let us say that is it right. And then in consume, I do P n chars I decrement the number of characters and I consume them consume an element. So, both of them are just decrementing; somebody also needs to increment it right. So, where do you increment?

Student: After (Refer Time: 29:49).

Right. So, I will increment here. What will our increment?

Student: N chars.

N chars right. So, you decrement n holes before you increment n chars after because you know at this point you have added a character and you can basically say you know you increment n chars by 1 and similarly, you can increment n holes here. That's it right? Semaphores go ahead.

Student: Do not we need mutex?

Do not we need mutex. Great question. So, that is it for making sure that you do not produce to a full q and it is also making sure that you do not produce from an empty q, but it does not make sure that these operations produce an element and consume an element are atomic with respect to each other and so, you may want to have another. So, you will want to have another thing we let us say you have an acquire mutex here just for the mutual exclusion part and then, you have a release mutex.

Student: Sir, should not acquire (Refer Time: 30:50) as an outside P and V?

All right let us see. So, the question is should not we acquire before P and release after V; what do you think? So, firstly, you know we said that P itself is atomic. So, it does not need to be provided protected by a mutex right and the. So, you know right this code is correct firstly, all right you need to convince yourself that this code is correct.

You have decremented n holes atomically and at this point you can be sure that you are free to consume or free to produce here right. Similarly, your decremented n chars you come out, so you are going to be sure that you are free to consume right. Mutex is just making sure that they are mutually exclusive with each other.

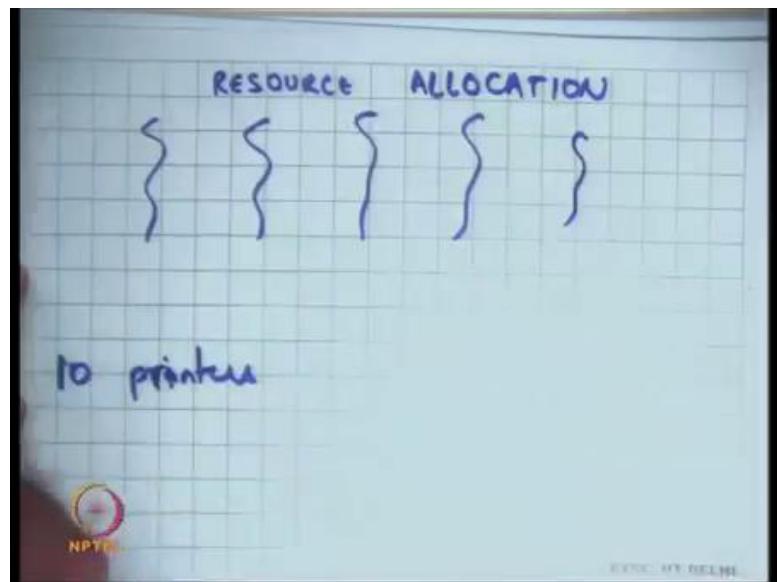
If you put acquire before P, you know you may actually run into problems because you know you here is the situation where you know that two things, there are two resources there is a mutex and there are n holes and you are going to, you want to acquire both of them at once and now, you have to worry about order and all that and so, you know you have to worry about deadlocks etcetera.

But if you do code in this way, then you know there is absolutely no problem of deadlock. Now, acquire and release a sort of mutex are within the inner loop and so, it can never happen that you wait on a mutex and then, you wait on something else. You are hold a mutex and you wait on something else that is not possible and so, you will never have a deadlock in this case right.

And of course, you also want to make your critical section as small as possible. So, that is you know that is producer consumer with some work with semaphores. This code is much simpler than the code that we had seen earlier right. What is going on? Well, what is going on as simple that you know we have noticed that there is a pattern, the pattern is that of counting; both of in both cases, we were counting, we were just counting the number of elements and we were waiting and so, we have subsumed that logic of counting within the P function itself right that is all we have done.

And so the our higher level code looks much simpler right and so, semaphores are a very useful abstraction, it allows us to capture this very common paradigm where you are counting the sources using P and V and your code actually looks much simpler and much better to understand, much easier to understand in that sense all right. So, this is an example of producer consumer with semaphores.

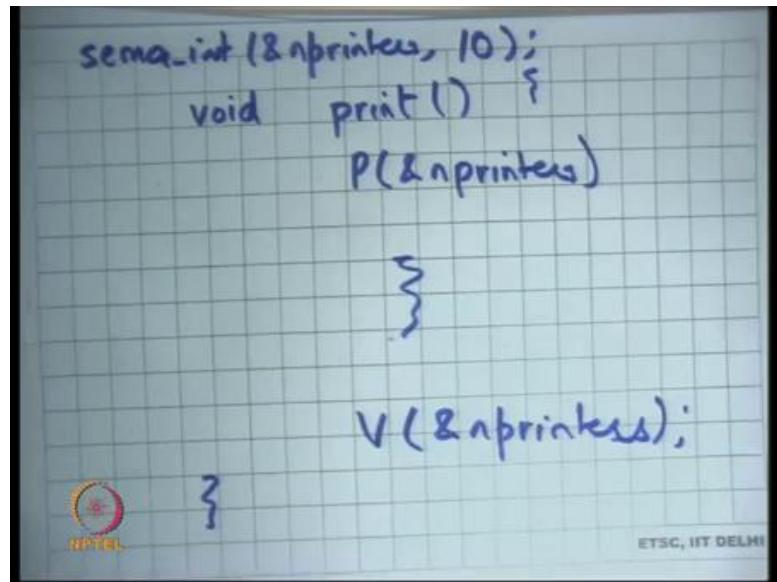
(Refer Slide Time: 33:24)



Let me give you some how another example. Let us say Resource Allocation all right. So, let us say you know I have 10 printers and there are you know many threads let us say 100s of threads that are trying to print on these 10 printers and I can I say that because there are only 10 printers, there should be utmost 10 concurrent requests in my printer, in my print q at any at any time let us say right. So, this is a problem of resource allocation.

You have multiple threads, you have hundreds of threads and a, but you have a few number of resources and you want to say that you know I want to give only, you know I want to make sure that the concurrency factor is limited to by the number of resources. So, let us say the resource is a printer, then utmost 10 concurrent print requests can be given, others should have to wait right.

(Refer Slide Time: 34:22)

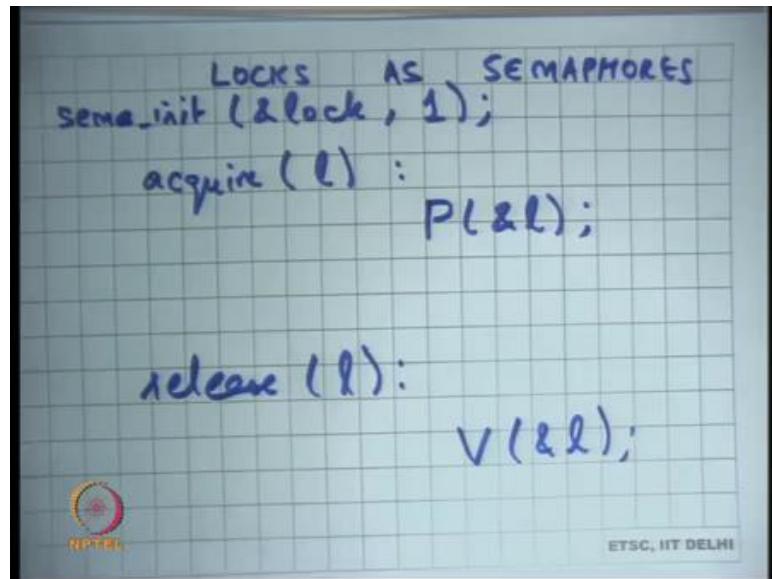


So, very easily handled by semaphores, all you need to do is let us say this is your print function, inside the print function you will basically say P on let us say n printers, you know call the print command and then, call V on n printers. Of course, you will first initialize the int printers variable semaphore to 10.

Let us say if you know you had only 10 printers or whatever is the number of printers you initialize the counter to the number of resources and then, whoever wants to consume the resource should or access the resource should enclose it is access logic within a P and a V and that is it right.

So, if there are a 100 threads depending on who came first, some of them will get it and as the threads are done now as threads exit they will immediately call V, which will cause another thread to enter right. So, it will allow you to do scheduling across and resource allocation across multiple threads all right. The semaphores for resource allocation all right.

(Refer Slide Time: 35:36)



And let us see another sort of interesting example is Locks, can be implement it as semaphore right simply. So, what is a lock abstraction? You basically have a state right. So, lock unlike condition variables are stateful is a stateful abstraction right. The condition variables are a stateless abstraction, semaphores are stateful abstraction and lock, it is also a stateful abstraction because the lock variable has the state called whether it is locked or not right.

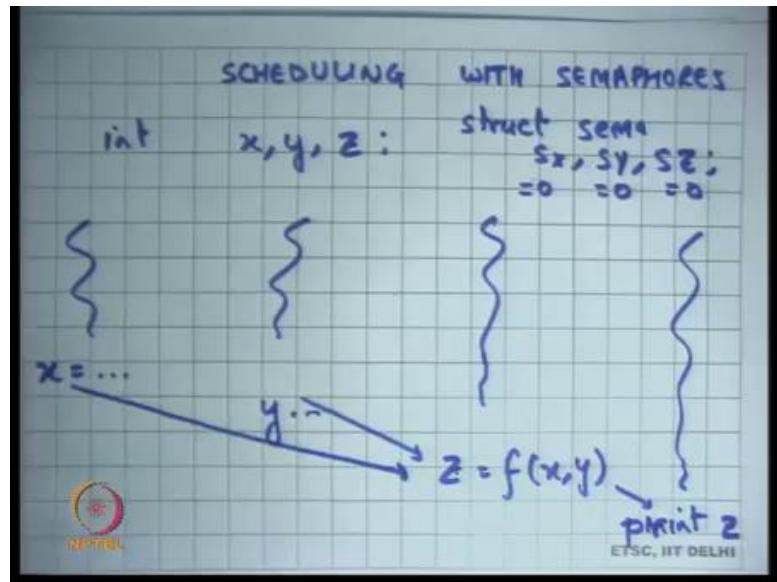
The lock variable has 1 bit of state the semaphore has an integer worth of state right ok. So, locks as semaphores. Well, what is a lock? It has acquire 1; well, you can implement acquire 1 by you know putting use having a semaphore inside 1 and you can just say you know you can you can say initial you can initialize the semaphore as 1 right.

So, a lock has done nothing but a resource with 1 instance and so, if the multiple threads we want to access this resource only 1 instance can go in it right. So, a lock can be model as a resource and you can use semaphores to do it. So, acquire 1 is nothing but what?

Student: P.

P. That is it and release 1 is V 1. Simplistic use of a semaphore you know semaphore who can do much more, but if you wanted to use the semaphore as a lock very easy we initialize it to 1, acquired becomes P and release becomes V.

(Refer Slide Time: 37:18)



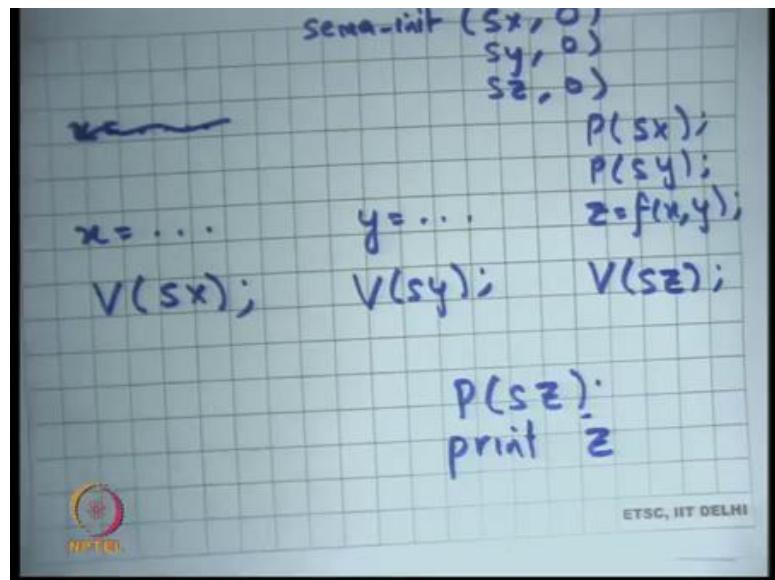
Let us see Scheduling with Semaphores. So, let us say; let us say there are you know these variables called x, y, z right. Let us see these are integers and somebody is going to compute x, somebody else is going to compute y and then, a third thread is going to use x and y to compute z and then, once z is computed you want to print the value of z right.

So, let us say there are you know 3 threads; one of them compute x, then another thread computes y, then yet another thread compute z as a function of x and y all right and let us say yet another thread, you know prints z. But you want to make sure that this computation happens only after this and this has happened, and you want to make sure that this print happens only after this has happened right. So, you want some kind of dependency.

So, let us say these are the dependency you want that you know z should happen after this has happened and this has happened, and this should happen after this has happened. So, this kind of some kind of dependency graph that you have and you are doing computation and different threads are going to compute this thing and you want to schedule these threads or you want to synchronize these threads in this way.

One way to do this is using semaphores. So, you can say struct sema sx sy and sz, you associate each us; 1 semaphore with each variable denoting whether that variable has been computed or not all right. So, sx and you initialize it to 0. So, all these semaphores I initialized to 0. So, you use you call the sema init function to initialize them to 0 and then, you write codes something like this.

(Refer Slide Time: 39:18)



So, when you say $x = \dots$ you say, so initially all of them are 0. So, $\text{sema_init}(sx, 0)$, $(sy, 0)$ and $(sz, 0)$ right. Then, you say $x = \dots$ and then you say what?

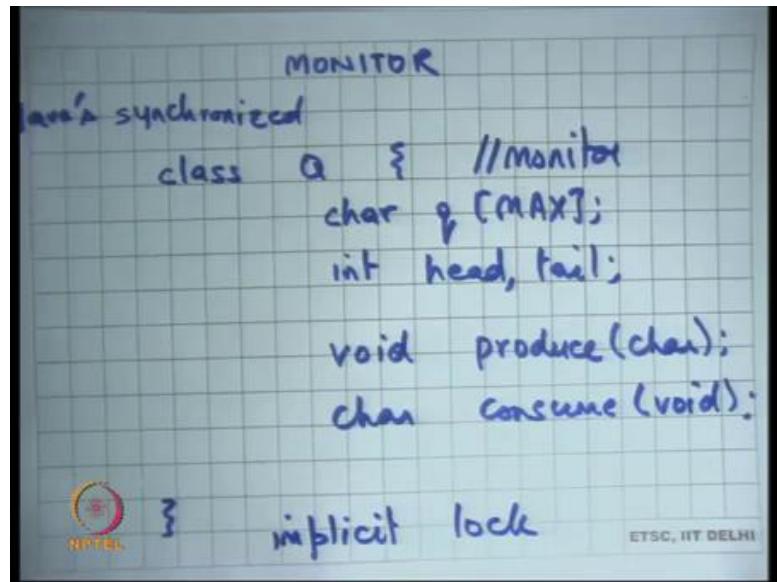
Student: (Refer Time: 39:40).

You say $V(sx)$ right and you say $y = \dots$ you say $V(sy)$. Basically, indicating that x has been computed. So, you put up your push up the flag by 1. Y has been computed, so you push a y 's flag by 1 and for $z = f(x, y)$ what do you do? Before it you say $P(sx)$ and $P(sy)$. So, P is going to have the effect of waiting for somebody to have called V .

So, only if somebody has called V would I ever be able to come out of P right because I initialize it to 0. Means so, this is an example of how you will do scheduling and then, after your computed z you may say $P(sz)$ $V(sz)$ and of course, in you know before the print z you may want to say $P(sz)$ right.

So, if you initialize the semaphore to 0 and use it to represent whether something has been computed or not or some condition is true or not, you can use that to make sure that things are happening one after another in a you know in an ordered way in a scheduled way, good.

(Refer Slide Time: 40:54)



Let me talk about another abstraction. So, we have seen 3 abstraction so far locks, conditioned variables and semaphores right and you have seen how you know locks are absolutely integral to do mutual exclusion and notice that these abstractions actually can be implemented, they are all actually roughly equally powerful abstractions. So, you can implement semaphores using locks and conditioned variables. You can implement locks using semaphores. So, you know it is not like one is less powerful than the other etcetera; but of course, you know that there is some difference.

Of course, you know you have to lock condition, you need both locks and condition variables to implement semaphores for example all right ok. Let me talk about something else now. So far you know all our abstractions have been of the type where there is some data that there is some data type that we declare like a lock or a semaphore or a condition variable and then, we have some functions and then it is the responsibility of the programmer to enclose his code or you know instrument his code with these functions appropriately. In general, it is very error prone.

This kind of thing is very error prone; you know he puts in acquire in the beginning and then, there are multiple code paths that are going from his function and then, now does he put a release on all those code paths or not. You know that is a very common bug, you have missed a putting a release somewhere for example, right.

Also, a compiler cannot check anything right. So, it is these abstractions are not part of the language per say. The C language has nothing to do with you know whether you are

using a lock or not using a lock whether or anything of that sort or you know so, these abstractions are completely independent of the language and that is not necessarily a good thing because the compiler will never be able to tell you if you made an error right.

So, example if you are did not enclose a you know if you are not bracketing and acquire with the release always, a compiler could have should have told you; but it will not be able to tell you because you know it knows nothing about what you are doing. For him, it is just function call called acquire right. You could have called it something else and you know you would not know.

So, there is a. So, to solve this problem, they there is yet there is an abstraction called a monitor which is a first-class support inside the language, inside the programming language to do a mutual exclusion right. So, the idea for monitor is basically that you know you basically use object orientation or etcetera and you basically define a class which will be called it is monitor and then, you find in you know and this class will have some share data.

Let us say in our case it has the shared data called q and these pointers head and tail and so on and it will have some functions which will operate on the share data all right. So, there is a function called let us say void produce right and char consume all right.

So, and it is the responsibility of the programming language to ensure that there is mutual exclusion between all the functions that are defined inside this class right. So, it is a regular object-oriented programming, there is encapsulation which means this these variables are private; only these functions can access these variables. But moreover, these functions will always execute in a mutually exclusive way.

So, if there is some thread which is executing produce no other thread will be able to execute either produce or consume right. So, only one thread could be inside the monitor at any time. So, this class is also called a monitor, a class of this type right is also called a monitor and the idea is that only one thread will be inside the monitor at any time right. So, it is a language that provides you a construct to define mutual exclusion. So far, we have been defining mutual exclusion using our own data type called lock.

Now, here is an abstraction that the high. This is the high-level abstraction provided by the language which allows you to basically say that these functions need to execute in a

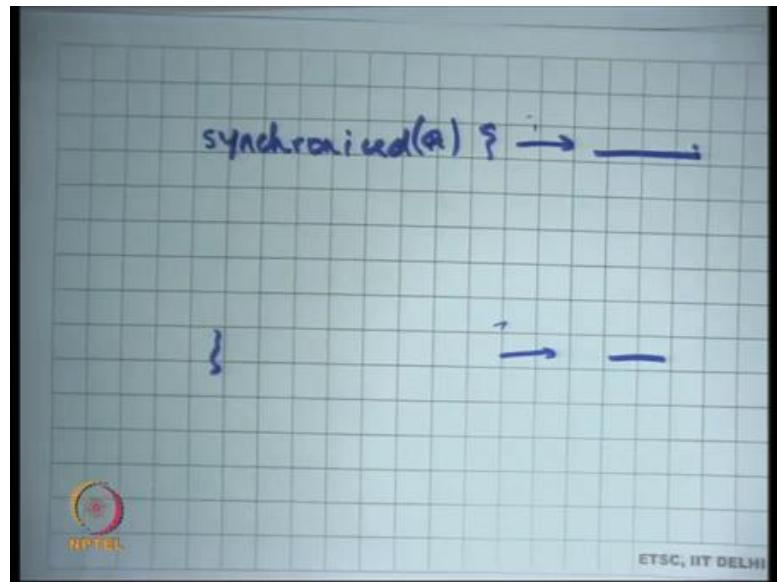
mutual exclusive way. It also provides you encapsulation because it defines the data which is likely to be shared and it says that these because these this data has only available to these functions, it automatically insures correctness because this data has not global in the true sense.

Then, you know it cannot because this data cannot be accessed by anybody else and because these functions are mutually exclusive by the definition of the monitor, you know your code should be correct. So, it makes things easy to reason about for a programmer, moreover the compiler can now you know has some idea about what this code is doing.

So, he knows that this is the data that is only being accessed by this, so it can do some optimisations in that sense right. It also knows that this function and this function are mutually exclusive. So, it does not need to worry about you know it can do it can freely to optimisations within produce and within consume because it knows that these accesses are going to be visually exclusive etcetera all right. So, for example, I could have implemented my q, the producer consumer my example using a monitor.

So, languages some languages support monitors, C does not support a monitor, but let us say Java supports monitor. So, the synchronised keywords. So, this is there is the keyword called synchronized in Java which allows you to say that this area needs to be mutually exclusive. It is the compiler which will put which is ensured that things are mutually exclusive for you fine. So, if you do not want to use explicit locks, you can just put code inside. You know you can.

(Refer Slide Time: 47:11)



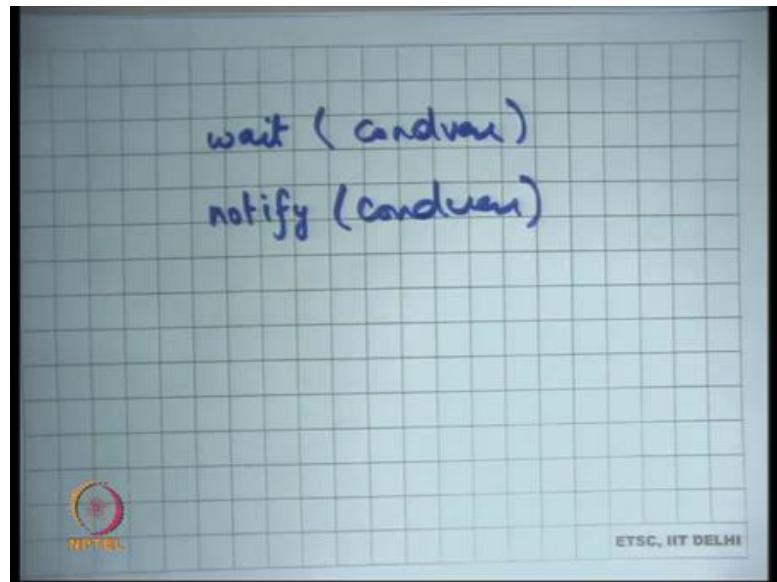
So, Java allows you to do things like you know a class is synchronised so that means the monitor or it can mean, or you can also say synchronised blocks. So, you can just say synchronised on you know and write some code here. The compiler will ensure that you know it basically amounts to saying acquired lock here and release the lock here and you can also give an argument to this.

So, this is the lock that gets acquired. So, that you know reduces the chances of you making errors like you know a release is not bracketed within an acquire. Now, because you are using static language construct, you are sure that you know you do not you are not running into errors of that type. So, that is a monitor ok. Basically, what the compiler does is that before and after the calls of these functions, it itself adds calls to acquire and release.

So, there is an implicit lock. In all our previous cases, there was an explicit lock that we had defined. Inside a monitor, there is an implicit lock that the compiler will insert for you right and also you know let us say there were multiple instances of this class q, then each of them will have the separate implicit lock for example, fine.

Monitors also need, so, the code within the produce just like you know we have seen the producer code. The producer code may need to wait on the condition. So, what is the; what is the counter part of a condition variable inside in the monitor world?

(Refer Slide Time: 48:50)



So, monitors also have these functions called wait and notify or you know you can call them by different names wait and signal or anything else and you can say some condition variable. Except that when you use wait and signal notify within a monitor, you do not need to use the mutex as a second argument of wait right.

The mutex of as it is which is the second argument of wait is also implicit if the wait has been called inside the code of a monitor. Then, it knows that this is the lock is basically the implicit lock that needs to be released right. So, the monitors wait and see notify do not need to care you know already know which is the mutex that needs to be released. It is the implicit lock of the monitor ok.

Let us stop here and continue our discussion tomorrow.

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

Lecture - 26
Transactions and lock-free primitives, read/write locks

Welcome to Operating Systems lecture 26, alright.

(Refer Slide Time: 00:29)



So, today I am going to talk about Transactions. So, you have seen locks as a way of providing atomicity and transactions are another way of providing atomicity right. What is atomicity? Atomicity is that you want certain piece of code to appear indivisible right, it should not be possible to interleave between instructions of that code and that code is called a critical section right. And, we use locks to basically you know we made an acquire before the critical section and made a release after the critical section and we said that that will ensure atomicity.

We also studied monitors, we said you know locks are error prone if there is a first class support in the language then it will become slightly easier for the programmer to reason about things and monitors was basically something that was also providing atomicity, but internally monitors are also using locks right. So, internally the monitors are also basically using an implicit monitor lock to basically provide mutual exclusion right.

So, how do the locks work? Locks work by, you know the programmer has to a priory tell that I am going to access this shared data and he tells it by calling this function called acquire and he calls it on this lock which corresponds for that shared data right. So, it is an a priory declaration by the programmer that I am going to access some shared data and this is the shared data I am going access and the semantics are that the acquirer will not let more than 2 threads access the shared data we will not let more than 2 threads fall right. This can be called a pessimistic way of doing controlling concurrency.

In the sense it is like saying that let us say I wanted to enter a room where I needed some privacy. I always whenever I enter the room, I put a lock inside it before I start doing anything right let us say you know whatever. So, I always put a lock irrespective of whether it is somebody else is actually going to try to enter the room or not right. In the if in the common case it is the case that nobody else is going to come into this room anyways right I am just doing this extra work of putting the lock first then doing my work and then unlocking it and then going out right.

But you know if nobody actually came during this duration while I was using the room you may say that this work is sort of wasteful. And, if this is the common case which in many cases it is right in many cases it you do not expect the probability that somebody will actually access the same resource at the same time is often small. So, you are saying that you are you know you are doing this extra work every time even though the probability of this event happening is small right. A more optimistic method would have been that I do not use a lot, I just enter the room and I start using it and if somebody comes later he opens the room and he tries to use that he figures out that oh somebody else is already using it and then he rolls back his execution.

In other words, he just you know in the physical analogy he just goes back and says you know let me ill retry some other time right now somebody else is using it. If I have that kind of an organization then in the common case when there is there is very little chance that somebody else is going to come at the same time you know I did not have to do any extra work to actually put the lock on the door I could just come in do my work and get out and if somebody else comes in during that time hell roll back if needed right.

This is more optimistic, this is more optimistic I call this optimistic, because I am optimistic that the probability that there will be a collision or there will be a concurrency

violation or atomicity violation is small right. So, this is more it is more optimistic in that sense plus locks are more pessimistic there you say that the you in sense you are some in some sense saying that the probability that somebody will actually violate atomicity is very large usually you are always basically doing this right.

So, in software this act of actually coming into the room checking if somebody just is using and if not and rolling back can be done by enclosing some the code into a transaction right. So, this whole process of actually chick coming in and trying to do something and then figuring out that you have failed because somebody else is using it and going back can be done by enclosing this entire code inside what is called a transaction right alright.

(Refer Slide Time: 04:51)

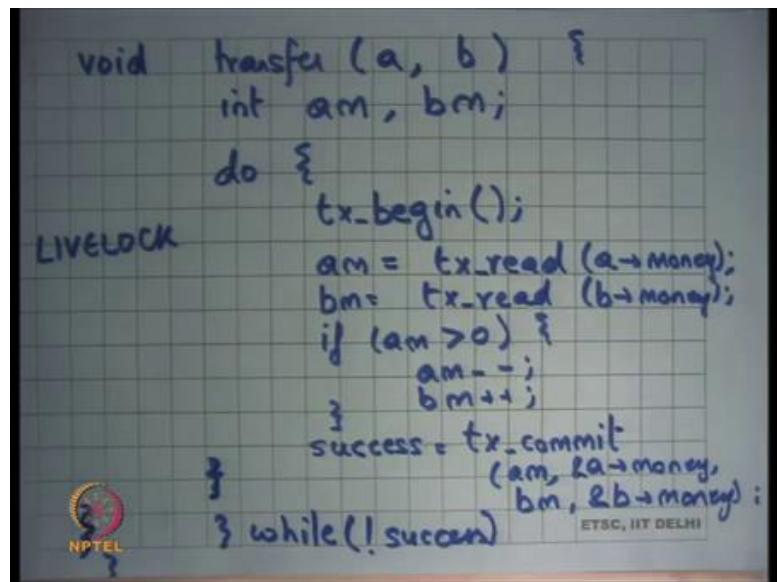
```
void transfer (account *a, account *b)
{
    if (a->id < b->id) {
        acquire (&a->mutex);
        acquire (&b->mutex);
    } else {
        acquire (&b->mutex);
        acquire (&a->mutex);
    }
    if (a->money > 0) {
        a->money--;
        b->money++;
    }
    release (&a->mutex);
    release (&b->mutex);
}
```

The image shows handwritten C code for a `transfer` function. The code uses fine-grained locking with mutexes associated with each account. It first acquires the mutex for the account with the smaller ID. Then, it checks if account A has money. If so, it decrements A's money and increments B's money. Finally, it releases both mutexes. The code is written on a grid background with an NPTEL logo at the bottom left and ETSC, IIT DELHI at the bottom right.

So, let us look at some examples of how what a transaction look. This example this running example that we have been using for transferring a unit of money from account to account b. And, here is the code which uses fine grained locking to ensure atomicity of this particular function right and we have seen this before.

So, you here is a mutex associated with a as a mutex associated with b and you basically take both mutexes before you do anything and then you release both. Once again this is a pessimistic I call this pessimistic because the probability that there is another thread that will access one of these two accounts a and b is very small right at the same time, yet I am pessimistically taking these locks each time right.

(Refer Slide Time: 05:45)



Let us see how I would have written this in the transactional way right. So, let us see I would have done something like this let us say void transfer from account a to b. I would declare 2 local variables am and bm rep representing a's money and b's money and I will use a do while loop to keep retrying let us. Let me write this code fully before we start discussing it `tx_begin()` we will say that a transaction has begun, am is a local variable which will read called `tx_read(a.money)` right and let us say `bm = tx_read(b.money)`.

And then you know so, what I have done is I said that there is a transaction; the transaction will first read the value shared value. So, `a.money` and `b.money` are shared values and I am reading those shared values into local variables right and then I am going to perform an operation on the local variables. For example, I will say if `am > 0` then `am--` and `bm++` right and finally, I will try to commit the transaction.

And, in the commit let us say I can give arguments would say right `a m` the value `am` to `a.money` and the value `bm` to `b.money` right and I do this while loop till I succeed alright. So, let us see what is happening. Firstly, I declared that what I am going to do, next is a transaction, by saying that everything that I am going to do next to the transaction. It basically means that whatever I am going to do is going to have a tentative effect, it is not going to be a final effect what I am going to do is tentative execution, it is like I am I have opened the room and I am trying to enter it. But, what I am doing it start tentative

execution and if I figured out that there is some collision somebody else was using the same room at the same time then you know I know where to go back.

I go back to the begin point right. So, that is what the tx begin is telling me that here is the transaction and what I am whatever I am going to do from here on is all tentative execution and it is going to be tentative execution till you commit it right. So, this code between the begin and the commit is all tentative execution and the commit function will atomically commit the results of the tentative execution on to concrete state right.

So, what am I doing is I read? So, I started a transaction then I said read the shared value into a local variable, read the shared value in the local variable perform some operation on the local variables and then try to commit the new values or the local variables to the shared variables, this commit function is atomic. So, commit function is atomic it will you know in one go try to update the values of a.money and b.money and it will succeed if between the tx_begin() and tx_commit() nobody else changed or read the value of these shared variables right.

So, I am basically saying here are the shared variables I have touched and here are the new values of these shared variables, commit them to the real state if nobody else has read or written these variables during this time. So, the question is what the problem is if somebody has only read it and I tried to commit it should commit fail if somebody has only read it.

Student: Yes.

Yes, it should right because it will again violate atomicity somebody has read the old value. So, atomicity means that whole thing appears as either has having done completely before it or completely after it right. So, if I have read let us say a before this transaction started, before this transaction started, I read a and then after this transaction committed, I read b, then atomicity is violated right.

Student: No, but what if let us say there is another function which just used the balance in the account that is left. So, why should I.

Right I mean. So, you are saying that you know it is possible that you may not need to commit, even if there was a conflicting read you may not need to fail the commit even if there was a conflicting read.

Student: Yeah.

That is true you know there may be some situations where that may be true, but in the general case that is not true. So, let us say there was another transaction that is doing the sum right. So, a running example is another transaction that is doing some right. So, let us say it reads a after that the transaction commits successfully and then it reads b that is a problem right. So, that is a problem because you there is no atomicity between the sum and the re transfer right.

So, you know you are right I mean in some cases you may be able to reason about it and say that you know you do not need to fail the commit if it is only a read because it does not matter my code does not you know semantics would not change it is still remains atomic. But, in the general case right because I mean in the general case you would want to say that if there was a conflicting read or a write then fail the commit and sum is an example in this case alright, ok.

So, it is going to commit it is going to try to commit it in an atomic way and if it tries to commit it and it figures out there was a conflicting read and write in this area while I was executing in this area then it will fail the commit right. So, how does it figure out this area? It just looks at you know it just looks at the time when the transaction was begun and between the commits. So, if in this time there was a conflicting read or write, it shows that there was a problem, you can optimize it further you know you can say that you know if there was a read or write here you may be no not clear etcetera.

But so at the, but at the very high level that is what it means that you basically look at what are the conflicting who are who are writing and reading what locations and then the commit operation is going to check this atomically and then either succeed or fail. If it fails then you just look back and you retry the transaction and you retry it a fresh completely a fresh right, you read new fresh values of a.money and you retry it again yeah there is a question.

Student: Means get a non-success or do one thing gets success.

So, do both sides get not success or does one thread get success it depends on the semantics of your transactional memory system you know either are possible, but let us say you know both of them get no success.

Student: But if both of them not get a success then they will get stuck in this (Refer Time: 13:17).

Right, so, depending on how you have implemented your transactional memory, it is possible you can implement semantics way or one of them is guaranteed to succeed which means there will always be progress and you know a simpler implementation and more efficient implementation may only guarantee we may not even guarantee that. So, it may say that you know it is possible that both of them fail right, in which case both of them are going to roll back and which has this problem of what is called a live lock right.

This is different from a deadlock so, it is called a live lock and live lock basically means that here are two transactions, that are continuously doing work, but none of them is actually making progress. In a deadlock none of the threads are doing any work and neither are they making progress, in a live lock threads are actually doing work, yet they are not making any progress. So, they just fall through try to commit both of them roll back and so on right.

And so, that is a really bad situation to have a live lock, but you know transactions are useful if you expect that concurrency is going to be real and so the probability of this live lock is very low right ok. So, all this sounds much more complicated perhaps than the locks yeah question ok, whenever concurrency occurs would not it always result in live lock, well I mean not necessary it is you know it is just a matter of what should you will happened.

So, in the first case it may happen that you know none of them succeeded, in the second case that may happen that one of them definitely succeeded, in you know the transaction memory system will have ways to tolerate a live lock one is that you know one will always succeed that is you know that is one way of doing it. The other is you know if there is a retry then you know maybe have some delay in the middle. So, there are multiple ways of making sure that live locks become less and less probable in the common case right. But, at in any case there is a lot of overhead to actually doing a roll back and retrying the whole thing again alright.

So, yes you can prevent real live locks in many ways one is you know make sure that your transaction system is basically always making progress at least one of them will successfully commit if there is a conflict or you know you basically make sure that retries are serialized you know the other ways that retries will always get serialized. So, you know if you figured out that something has failed then you try to serialize execution yourself you know. So, those are all ways to do things make sure that live lock does not happen here yeah.

So, what is the advantage of such an implementation I said that no locks have this extra overhead of actually locking and you know it would have been better if I had been more optimistic, but this looks even more complicated right. Why is it more complicated or why is it more expensive I should say? Firstly, we have to roll back, but let us say you know rolling back is a very rare operation because I started with the assumption that the probability of concurrent access to a shared resource is small right.

So, let us say the probability of a rollback is very real right and that is true in many cases, that is true for the account example the transfer example that we have taken right the bank account example, so anything else.

Student: Data.

Exactly so, you have to basically track what are the locations that have been read, what are the locations that have you know and access and so on and you know how to do this bookkeeping somewhere right and you know how do you do this bookkeeping. You basically you know for this for every memory access for every memory location and for every memory access you have to store some look at some data that is saying you know whether it was accessed or not and these kinds of things and this seems very expensive, this seems much more expensive than actually doing taking just a lock right. A lock just involved setting a bit 2 0 or 1 right.

So, yes, it is expensive because you need to do extra bookkeeping about what memory locations are accessed. Transactions are or you know what resources are accessed, I should say instead of memory locations because memory location is one of one type of resource, but there could be other types of resources. Transactions are used widely in databases or you know anything that involves disk accesses file systems for example, why because it is easy. So, each read operation in transactions that involve disk

operations involves a disk access a disc access or a disk block access or disc access can only be done in granularity of a block right.

So, what would take a you know you either read a block or write a block and you at disk access is very expensive right it is already milliseconds to access a disk. And so, this extra bookkeeping information about which blocks have been accessed and which blocks have not been accessed is very small in comparison. It can be stored in memory and it is very efficient to store this in memory it is not a big deal right. So, transactions are very useful if this operates the read operations or the access operations themselves are very expensive right, then this extra overhead of actually doing this bookkeeping becomes very small relatively ok.

So, transactions are great for something like databases and they are used in fact, to ensure atomicity of database transactions. The other advantage of transactions is that you do not have to worry about a lot of things, you do not have to worry about fine grained locking, you can put a large area of code into one transaction and it is the runtime system that is keeping track of what you have read and what you have written and whether you want to roll back or not right. So, for example, notice that in this code nowhere am I associating a log per account or anything of that sort.

I have just put the entire transfer function within a within one transaction and it is a run time system that is figuring out whether this transfer and that transfer conflict or not right, by look comparing the read write sets you know what memory locations has he touched or what resources has he touched. And, what resources has he touched I am just comparing it as a run time system that is doing it the job of the programmer becomes much simpler right.

So, if he does not have to do fine grained locking he can put a transaction around the entire code, he does not have to worry about you know figuring out the right locks are placing them in the right place, he does not have to worry about deadlocks that is a big one right. Deadlocks as we have discussed is a big problem because firstly, you are to have a global order on all the locks that is often very difficult to get.

You have to reason about you know what are all the locks that different areas of your code have, different types of locks etcetera and all of them have to be in a global order that is a really a difficult thing to do and it also kills modularity. As we have seen before

it does not allow you to write your code in a modular fashion, because now you have to worry about what kind of locks is this he is going to take or he is going to take etcetera and what operations do I need to be an atomic etcetera right, with transactions anything that I need to be atomic I just put a begin transaction and an end transaction around it and we done it ok.

So, it is very useful in that sense and that is one of the primary reason that transactions have actually been very successful in the database community or database world, because you know having to do locking in databases is actually very very error prone. Apart from being expensive the bigger problem were doing locking in databases it is very very error prone you have to worry about what are all the let us say tables in your database and whether you know you are taking the locks in the right order and things like that. On the other hand, you just put it in a transaction and the database system, or the runtime system will figure it out for you alright.

Student: Sir

Yes.

Student: Sir we have said that commit is atomic. So, for ensuring the atomicity do not we require locks.

Right. So, how commit is implemented, is another thing. So, really depends on your run time system how you will implement commit and the question is do not I require locks to implement commit, you may or may not depending on you know let us say you are doing your two implementing transactions on disc locks, yes you require locks. But these locks are only named in memory locks right and more importantly so and more importantly the critical section is very small of your lock it is just about updating the memory right there right.

So, I mean even if you require locks in the commit it does not you know it does not obviate the advantages of or it does not reduce the advantages of transactions that is the problem if it requires locks right. The programmer still does not have to worry about things right and for databases it is still you know equally expensive or less expensive than using blocks right. Now the question is, do transactions make sense in operating systems? Alright.

So, what is the difference between operating systems and operating systems, these are likely to be memory accesses instead of this block accesses and so, this business of actually doing the bookkeeping has is actually you know bookkeep doing the bookkeeping is likely to become more expensive than actually doing the access right, because you have to do at a bookkeeping on every shared access that you are doing. Also the commit is likely to become very expensive because as I was pointing out you will probably need locks and you know and then needing having to go through all the read list and write lists and performing intersections.

So, for that reason you may say that transactions are not really suitable for operating systems and that is what, that is the reason that so far at least in our transactions are not usually used in operating systems and because you know they are too expensive to be used, lock seem to be cheaper to use an operating system. However, you know given that we are moving in the, we are moving towards multiprocessors and more and more processes and so on.

There is a huge need to make more and more of our code more and more parallel and that has huge software engineering challenges for programmers, to be able to do fine grain locking and all this and it makes your coop program very very highly non-modular right. So, later you know very recent processors like the Intel's Haswell processor actually have hardware support for transactional memory.

So, the idea is that they have instructions called transaction begin and transaction commit alright and using those instructions it is a hardware that will do all the bookkeeping for you within that block of transaction begin and transaction commit. And, when you do the transaction commit it is a hardware that will compare the read and write lists of the two transaction and decide whether the commit was a success or failure right.

The assumption here is that doing these things in hardware is likely to be much faster than having to do these things in software alright and there is evidence to believe that at least for some number of processors it is advantages so, to use transactional memory over locks right. And so, that is the reason that a lot of investment has gone from many companies and you know one example is Intel's Haswell processor which is which is which is actually being available today alright ok.

Student: (Refer Time: 24:42) Hardware support is for (Refer Time: 24:45) transactions on disk lock access or.

No, the hardware support is for transactions on memory accesses alright. So, every so, if you say a transaction begin and then you make a memory access it is a hardware that will do the bookkeeping about what memory accesses you have read or written and when you do a commit it is a hardware that will compare these lists of memory accesses and decide whether the commit is successful or failure right.

And, so that is you know that is the domain of transactional memory I would not go into too much detail about it, but just to just to tell you about transactions how they have been so successful in databases and how they have been very seriously looked upon as a potential option of handling concurrency in operating systems by using hardware support alright.

Student: Sir.

Yes questions.

Students: To maintaining the modular I think cannot we use conditioner locks such that if it is taking the a function and making a sir function call function call and it (Refer Time: 25:47) the lock and give the lock to the calling function.

So the question is you know I am saying that locks actually hamper modularity and I am saying that because if there is a function that takes a log and then he calls another function that takes another log then there is a possibility of deadlock and all that and the question is if you take one another lock after holding one lock cannot you do something called a conditional lock were the second lock basically is least if you take the second first the first lock is released before you take the second log what do you think.

Student: But the caller would not reach the (Refer Time: 26:21).

I mean if you can just release the lock arbitrarily right I mean the lock is there for a reason you cannot just I mean if you just started releasing locks then you are killing then you are then you are losing atomicity.

Student: But when the function that, sir the function that we are calling it at the same time acquiring a same lock right.

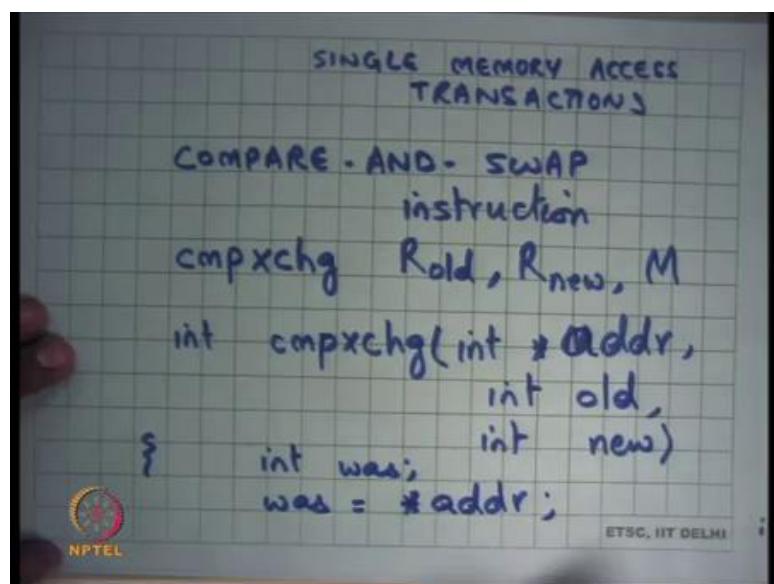
It is not acquiring the same lock it is you know so, modularity is killed because it may be acquiring a different lock right and then you have to worry about the order of these two different locks and I mean if it is the same lock then you know what you are saying is similar to recursive locks that we have discussed.

Student: Yeah.

But and we also discussed problems with recursive locks in any case recursive locks are an option, but really I am not talking about modularity being hampered because of acquisition of the same lock I am talking about modularity being hampered because of acquisition of different locks, because locks need to be acquired in a certain order alright

So, basically transactions are an interesting idea people are looking at it and whether they can be used in operating systems or not remains to be same. However, one type of transaction is actually being used for many years now in operating systems and these are transactions that involve single memory access right.

(Refer Slide Time: 27:41)



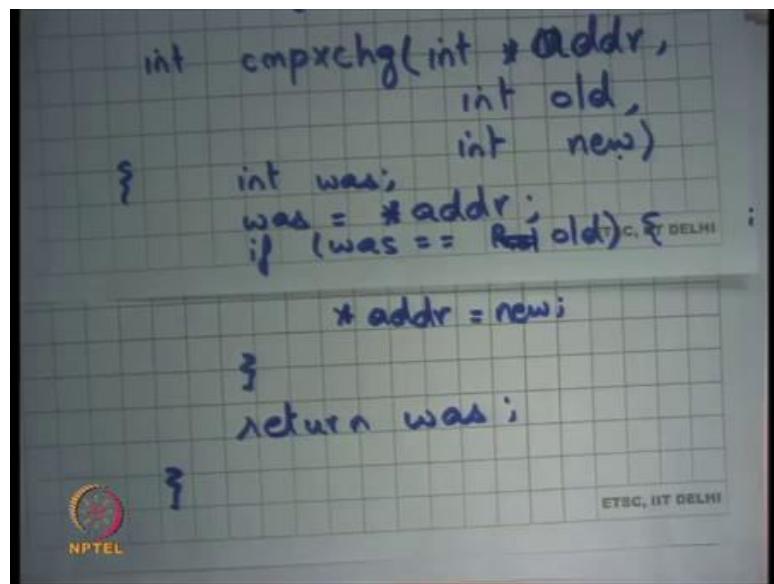
So, if you can write your code as a transaction such that it is going to access only one shared memory location and then you know it is then the hardware has been providing support for a long time to implement such a single memory access transaction. And,

operating systems have been using it for a long time operating systems and in general concurrent programs have been using it for a long time to do this and one common paradigm to do this is basically what is called a compare - and - swap instructions right.

So, let us see what this compare, and swap instruction is if I was to right the c semantic. So, compare and swap on x 86 is called the compare and exchange instruction, it takes 3 arguments register one which I am going to call R old let us say, just say 2 I am going to call R new and a memory location M right. And, the semantics of this instruction are the following I am going to write C code to describe the semantics let us you know let us say I write it as a function then it is compare and swap and star M or let us say address just to make it clear int old and int new right.

And it just no first reads let us say was is equal to star. So, it reads the value in addr into a local variable or locally into the in the hardware for the instruction.

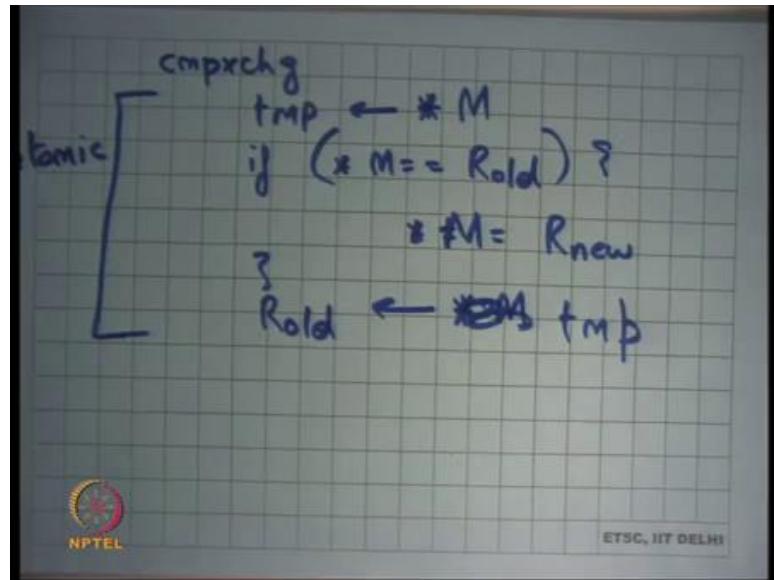
(Refer Slide Time: 29:57)



And, compares the current value with R old or with old let us say in the c things, let us see if it was equal old then $*addr = new$ and return was alright I am going to explain it very soon ok. So, what is going on? Basically the semantics are check the contents at location addr, compare them with location old with value old right, check the contents location addr, compare the contents with value old, if they are equal then replace the contents of addr with new otherwise do not do anything right.

So, the idea is I have previously read some value from this location called addr, I have it in a register. Now, this instruction is going to atomically check if the value of that location is still the same and if so, then it is going to replace it with the new value and if it is not still the same then it is not going to replace it with the new value right.

(Refer Slide Time: 31:29)



So, let me write it in plain English first, compare exchange if $*M = R_{old}$ then $*M = R_{new}$ that is all in some sense right and this is done in an atomic fashion, you check if the value in the at that address is the same as the value that you have in your resistor. And if so, then you replace it with this new value otherwise you do not do anything let us say and just one more thing it basically also returns the old value of $*M$ or let us say let us say temp is $*M$ it returns the old value of $*M$ in R_{old} .

This operation is actually very similar to the semantics of the commit operation in a transaction right. What is the committee operation? The committee operation says change the value of these locations with these new values if nobody has written in the middle right. So, they compare an exchange in suction is the same similar, change the value of this location to this new value if it is value is the same as the value that I have read before right.

So, there is nobody has written in the middle is captured by the value is the same right and this is an atomic instruction you know you can put the lock prefix behind it to make it atomic.

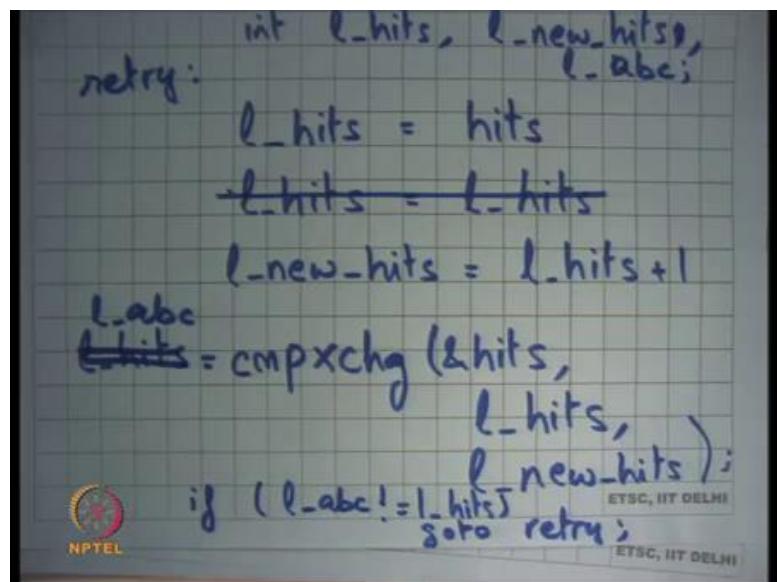
So, what am I going to do? I am going to use this instruction to commit one memory access to transactions. So, any transaction that can be that has done only that needs to do only one memory access, one shared memory access can be modelled as a transaction. So, any atomic region, any critical section that can be modelled as one update to a shared memory access region can be modelled as a transaction using the compare and exchange instruction right ok, let us see these examples.

(Refer Slide Time: 33:59)



So, let us say I had this operation where I wanted to do hits is equal to hits plus 1 right. So far, I was using locks to do atomicity of the code right. So, I was basically saying acquire here and release here right, now let us try to do this using it transactions right. So, I can say so, if I want to do a $\text{hits} = \text{hits} + 1$ and I want to do it in an atomic way I could do something like this I could say.

(Refer Slide Time: 34:33)



Let us say I declare a local variable called local hits and I will declare another local variable called local new_hits let us say just 2 local variables it does not matter how many local variables are declared and I read the value of hits into local hits right. Then I do this operation hits = hits + 1 well let us say sorry let us I do l_new_hits = l_hits + 1 ok.

Now, what have I done? I have read the old value of this shared variable hits in a local variable, I have performed some computation on this local variable and now I have both the old value of that shared variable and what are the new value that I want to put in the shared variable. And so, what I am going to do is, I am going to say compare exchange what, hits as the address, l underscore hits is the old value and l underscore new hits is the new value alright.

So, I am going to atomically try to update the shared variable hits with l_new_hits, but I will only do it if the hits variable still is equal to the old value. If it is not equal to the old value what does it mean? Somebody has concurrently tried to increment hits and so, you should not commit you should not update now right. So, you should not have the lost increment problem right.

So, this will you know this will return the old value of hits whatever the value was read into hits and let us say it is returned and register I want to write it in the functional form. So, l underscore hits is equal to compare an exchange this and I basically say if or let us

say what let us say let us say I say let us say I declared another variable called local. Let us say local abc I do not I cannot so, let us say I just say l_abc is equal to compare exchange this. So, this is going to return the value that was read by this atomic instruction and recall that this entire operation of reading the value, comparing it, and updating it is completely atomic right. So, I basically read the old value into l_abc and I say that if $l_abc \neq l_hits$, then what do I do.

Student: Retry.

Retry right so go to retry, now let us say retry is going to be where right you can write it in loop form I am just using go to because of lack of space, but you know basically you just want to retry the transaction you can write it as a wild loop for example, also it does not matter. So, this code is ensuring the atomicity of hits++, but in a transactional way ok, also this transaction guarantees that at least one transaction will always succeed it guarantees progress.

So, this transactional system so, we had this discussion on if there was a conflict or if there was a failure at commit do both roll back or does or one of them will always succeed, in this case one of them will always succeed right. Whoever executed the compare and exchange first and notice that the comparing changes an atomic instruction is going to succeed the second one is going to fail, and he is going to go to retry.

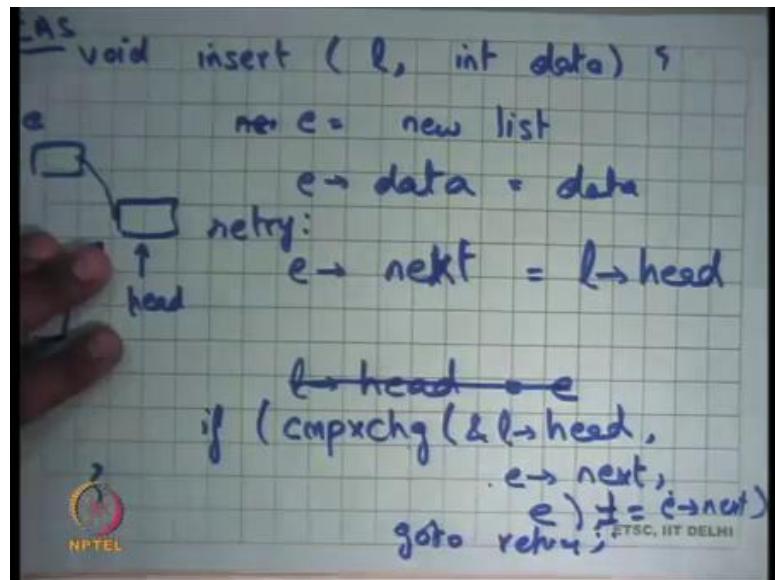
Student: If anybody else reads it would not fail.

Yeah, it will allow multiple reads, so it is basically guaranteeing atomicity against concurrent rights. In fact, it is even weaker than that it is not guaranteeing atomicity against concurrent rights it is guaranteeing atomicity against concurrent changes, if a concurrent right happened, but it left the same value it does not matter right. So, it is not a complete transaction in the strict I mean it is not the same semantics that we discussed it is not comparing the read write sets. It is actually looking at the value and if the value has changed then it will fail if the value has not changed then it will succeed.

So, if there was a concurrent right that you know that wrote it, but it remained the value remain same no problem right. So, that is the semantic of (Refer Time: 39:12) and whether it fits in your logic program logic or not that is really up to you to decide ok. So, notice that I have avoided locks in doing so, and so all the problems associated with

locks are also gone in some sense. Firstly, I do not need an extra shared variable which is a lock and secondly, you know because there is progress guaranteed now there is no problem of deadlock alright. Now, let us take another example we were looking at this insert function right initially.

(Refer Slide Time: 39:43)



So, we said insert into list l some data right and we had this lock we had this peerless lock to do this insertion and let us try to write this insert as a transaction right. So, how let us see I mean using comparison swap right. So, comparing swap is also called CAS and I am going to use the word CAS to represent compare and swap instruction or compare in exchange right, it is a common term called CAS to use to basically do this right.

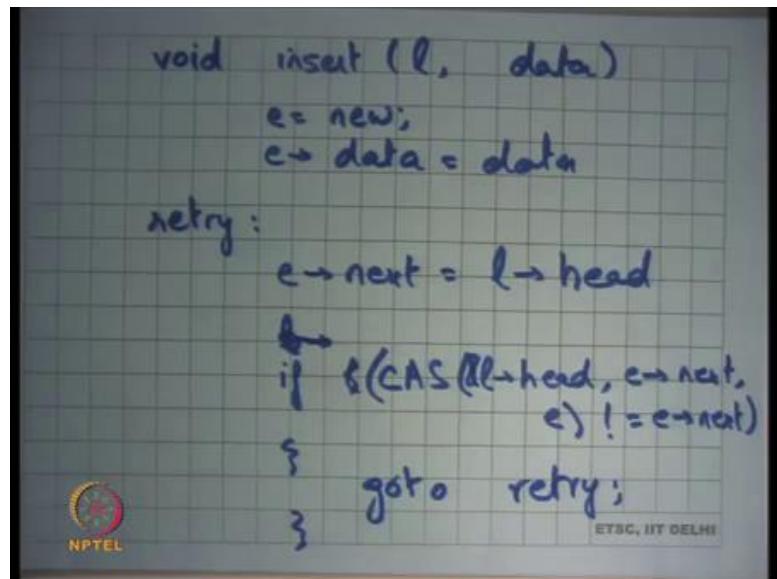
So, how can I do this, well I can say let us see new let us say e is equal to new list, e.data = data and then I say e.next = list.head right and finally, I am going to say list dot head is equally right recall that was my code list dot head is equal to e right. So, notice that all these 3 locations or 3 instructions are only operating on local data or at least read writing to local data they are not writing to shared data.

It is only this final instruction that is writing to share data right and it is possible to write this using CAS without having to use locks in the following way, you can say you can do compare an exchange on l.head with old value being e.next right and the new value being e good right. So, you atomically try so, what is happening let us say this was head you

have already made e and you have made it point to head and now you atomically try and you already know that you know the time when you read 1 dot head, head was here. Now, you automatically tried to make head here, but you only do that if head is still equal to this. If there was a concurrent insert, then head would have become something else right if there was a concrete insert head would have gone here.

So, let us say there is a concrete insert and somebody was trying to do this and so, let us say he won then head would have come here and so, the second insert would have failed right. So, if this succeeds that basically means that nobody has been able to do a concrete insert for me and so, what I do is, if you know if compare exchange is equal to e dot next. It is not equal to e dot next then retry right so, go to retry, and retry is again, where should retry b yeah just before here right let me write it again.

(Refer Slide Time: 43:01)

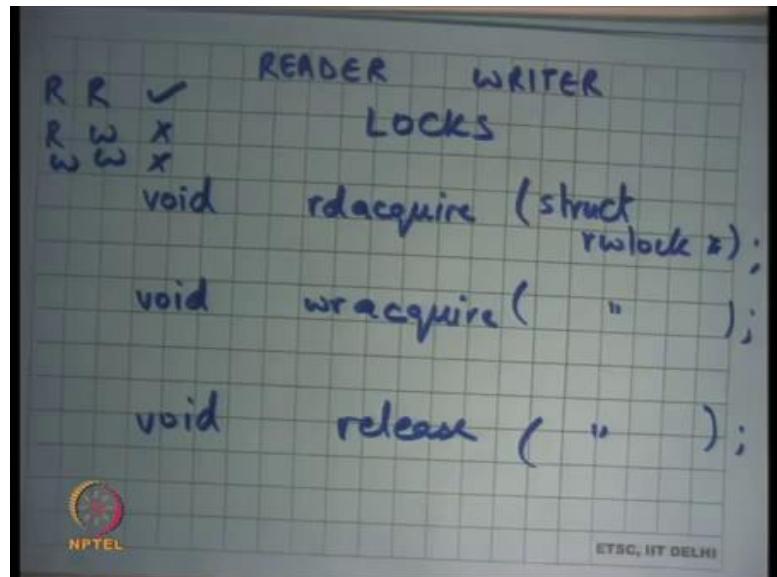


void insert (l, data) e = new; e.data = data, retry e.next = l.head; l.head or now this is the this is the tricky part if CAS instead of compare and exchange I will just use the word CAS if (CAS l.head actually &a.head, e.next, e) != e.next then go to retry ok. So, this is a lock free way of ensuring the toxicity of the insert operation alright.

So, these are these are called lottery operations it is compare and swap kind of things are called lock free operations and you know at the heart of it is really a transaction with a single memory location alright, with the different kinds of semantics it is not doing read write set intersections it is actually looking at the value to decide whether there was a

conflict or not. Finally, so, any questions on transactions before I move to my next topic very briefly, alright good.

(Refer Slide Time: 44:51)



So, finally, I like to point out that there is something called a reader writer lock right, the idea is that often they are multiple functions, some of the functions are only interested in reading the shared memory location or shared object and some location some functions are interested in also writing to the shared memory location.

It should be possible for multiple threads to concurrently read because you know one read does not you know adversely affect another read and it is only a write that effects adversely effects a read or it is a only a write that adversely affect a write right. So, the conflicts are only between read writes reads and writes or writes and writes there is never a conflict between a read and read.

So, if you expect that your code has lots of readers and very few writers then it is possible to use this change of fraction called reader writer locks. And, it has 3 types of functions locks have acquire and release these have read acquire and read release and write acquire and release right same struct lock, let us say struct r w lock right. So, the idea is that the fraction now says that multiple threads can hold the lock in read mode simultaneously or the lock can be held in the read mode simultaneously, but if somebody hold, but a lock can only be held once in write mode alright.

So, acquire basically made sure that only one the lock can only be acquired once at one time right it cannot lock cannot be acquired by 2 threads simultaneously. In this case a lock can be acquired simultaneously by 2 threads in read mode that is possible. But if, but it is not possible to have a lock being acquired in by 2 threads in write mode, it is also not possible for a lock being acquired in a write mode and the read mode simultaneously right.

So, in other words read and read, read is allowed, but read write or write, write is not allowed ok. So, that allows you to have more conquering a system if you expect that they are going to be lots of readers and very few writers and first alright good.

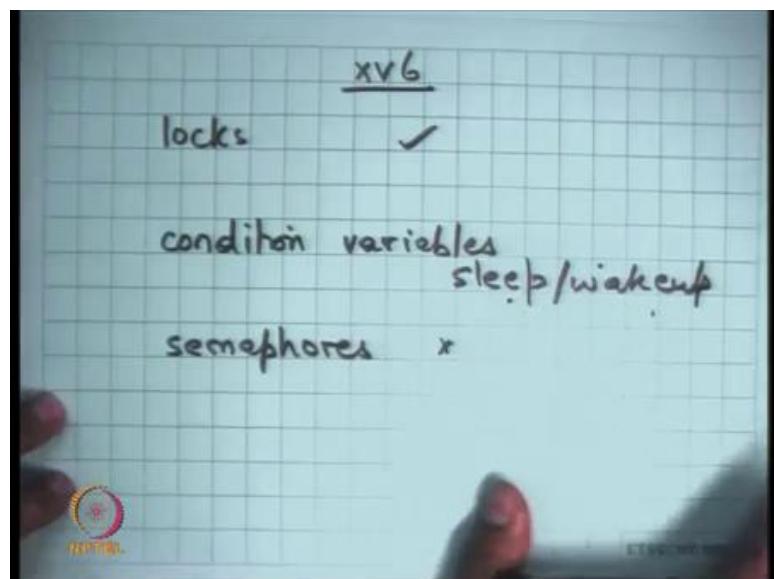
So, let us stop.

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

Lecture – 27
Synchronization in xv6: acquire/release, sleep/wakeup, exit/wait

Welcome to Operating Systems lecture 27 all right. So far, we have looked at locks condition variables semaphores, we also looked at lock free methods like transactions and one manifestation of the act called compare and swap right. So, these compare and swap instructions are also what are called lock free methods of dealing with synchronization and their advantages of doing lock free synchronization as we are going to discuss later in this course right.

(Refer Slide Time: 00:31)

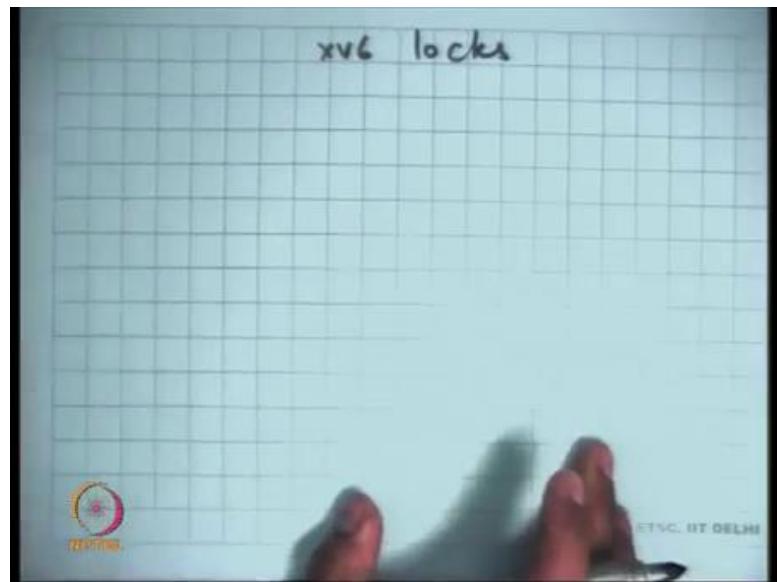


So, today I am going to discuss these in the context of xv6 and let us look at how an operating system like xv6 is using synchronization under it within itself. And so xv6 uses locks xv6 uses condition variables but you know in a different form. So, instead of calling them condition variables you call them sleep wake up all right. So, the semantics are similar sleep is conditional wait.

Student: Ok.

And wake up is notify. So, instead of calling them instead of calling them condition variables and wait and notify we calls them sleep and wake up. And we are going to see what the subtle differences between sleep are and wake up and wait and notify and does not use; does not use semaphores. So, that is fine semaphores I anyway (Refer Time: 01:42) you can always simulate anything that you need with semaphores using locks and condition variables all right.

(Refer Slide Time: 01:57)



So, let us look at how the acquire function is implemented in so let us look at xv6 locks all right. So, let us look at how xv6 is implementing locks. So, this is sheet fourteen on your listing ok.

(Refer Slide Time: 02:07)

```
1464 lk->name = name;
1465 lk->locked = 0;
1466 lk->cpu = 0;
1467 }
1468
1469 // Acquire the lock.
1470 // Loops (spins) until the lock is acquired.
1471 // Holding a lock for a long time may cause
1472 // other CPUs to waste time spinning to acquire it.
1473 void
1474 acquire(struct spinlock *lk)
1475 {
1476     pushcli(); // disable interrupts to avoid deadlock.
1477     if(holding(lk))
1478         panic("acquire");
1479
1480     // The xchg is atomic.
1481     // It also serializes, so that reads after acquire are not
1482     // reordered before it.
1483     while(xchg(&lk->locked, 1) != 0)
1484     ;
1485
1486     // Record info about lock acquisition for debugging.
1487     lk->cpu = cpu;
1488     getcallerpcs(&lk, lk->pcs);
1489
1490 }
```

That is the lock function, that is the acquire function basically, let see what the lock structure is before that. So, lock structure you know if you want to initialize a lock you basically have three fields in the lock one is the name, name is just for debugging purposes. If you want to know what lock it is so it is just for debugging purposes you do not really need, but they are using it for debugging.

Then there is this lock variable that we know about that is the state of the lock and then also which CPU is holding it that is also again only for debugging you do not really need it strictly, but that is you know it helps in understanding what is going on all right.

(Refer Slide Time: 02:42)

```
1469 // Acquire the lock.  
1470 // Loops (spins) until the lock is acquired.  
1471 // Holding a lock for a long time may cause  
1472 // other CPUs to waste time spinning to acquire it.  
1473 void  
1474 acquire(struct spinlock *lk)  
1475 {  
1476     pushcli(); // disable interrupts to avoid deadlock.  
1477     if(holding(lk))  
1478         panic("acquire");  
1479  
1480     // The xchg is atomic.  
1481     // It also serializes, so that reads after acquire are not  
1482     // reordered before it.  
1483     while(xchg(&lk->locked, 1) != 0)  
1484         ;  
1485  
1486     // Record info about lock acquisition for debugging.  
1487     lk->cpu = cpu;  
1488     getcallerpcs(&lk, lk->pcs);  
1489 }  
1490  
1491  
1492  
1493 XV6  
1494  
1495
```

So, this is the acquire function it you know it disables interrupts. So, pushcli is going to disable the interrupts on the current processor. So, notice that the acquire function disables interrupts uses this loop to atomically set the locked variable to 1, sets the CPU to the current CPU value and the this is some debugging function which allows you to log exactly who called this lock and that is it.

So, what does it mean, when you get out of acquire these interrupts are still disabled right, because you called pushcli and you never called pop right you never enable re enabled interrupts? So, for the entire critical section in this spin lock the interrupts are disabled. So, why does xv6 need to disable interrupts in the entire critical section?

So, one answer is that if this timer interrupt within the critical section, then you know it can get soaked out and somebody else can get to run and atomicity you can get violated is that right. No because you know the other thread should also be taking the lock and it will not get the lock, because it has already called the exchange interaction here right.

Student: Ha.

So, it has set the lock variable to 0 to 1. So, any other thread will not be able to set you know we able to acquire the lock.

Student: Because esters and interrupt handler then basically. So, it may try to reacquire a lock, so it was acquired initially by requirement.

Right, so what if the interrupt handler also tries to access your data right. So, we have within the kernel and it is possible that whatever data you are accessing the interrupt handler also tries to access the same data all right. So, it is not about multiple threads, recall that we have been talking about mutual exclusion across threads.

Mutual exclusion across threads it is easily handled by this atomic exchange instruction right on multiple CPU. But what about mutual exclusion between a thread and the interrupt handler right. If an interrupt gets to run in the middle of a critical section then and the interrupt handler could touch the data that you have in the middle of a critical section, then we need a way to protect it right.

So, what are some ways to protect it you could say let the you know would not the interrupt handler also try to acquire the lock right. But if the interrupt handler tries to acquire the lock then you have a deadlock right there right. So, you got an interrupt, the interrupt handler gets to run and then interrupt handler tries to acquire the lock that is already held right.

So, that is not a good thing and interrupt handler will likely run with interrupt it saver, so that is not a good thing now you basically have a deadlock. So, basically to protect a thread from the interrupt handler from concurrent accesses by the interrupt handler, you disable the interrupts for the entire critical section.

The reason this is acceptable in xv6 is because you will expect that the critical sections are small and you know critical sections that are protected using this spinlock are small and those critical sections are indeed you know need to be protected against accessed from interrupts and we are going to see some examples right.

In general when you write your programs you would not worry about, so let say you have your own program or you know you write a own kernel thread and that thread has nothing to do with an interrupt handler. Then you know you will not use xv6 spinlock you would probably want to use your own spinlock that does not disable interrupts because you do not care about atomicity with respect to the interrupt handler.

So, the interrupts are disabled for the entire length of the critical section acquire leaves the interrupts disabled.

Why do I need to pushcli why cannot I just use cli? Just to ensure that if there are you know nested calls to acquire the nested locks. So, you acquired lock one and then you acquired lock two then you have a count of how many times cli was called right and so that is many times you have to call cli before you actually re enable the interrupts.

So, if you have you know nested locks then that is why you know. So, pushcli is nothing but it just increments a counter it is perceive counter that it increments, basically saying how many times cli has been called all right. So, that is acquire and let us look at the other things. So, notice that this holding function what is it doing it just checking that whether I am holding the lock already, whether this CPU is holding the lock already.

So, this is just a debugging thing really and the reason this is you can check this is because you also have a field called CPU right. So, holding is just going to check that you know if the same CPU try to acquire the same lock twice that is a bug and you want to you know just tell the programmer right away that there is bug in your program right rather than a (Refer Time: 07:38) have to find it in a roundabout way.

So, this is just a debugging aid right in general if you did not have the CPU field you won't have been able to make this check and that is program if you written your code correctly that still correct but this is a debugging aid for you.

Student: Sir.

Right, so I mean the other semantics in this sort of I mean this slightly subtle semantics are that a lock is actually held by a CPU right. So, I mean that falls out from the fact that you would disable interrupt. So, a thread and a CPU basically mean the same thing, because if disable the interrupt. So, the if you disable the interrupt the thread is now stuck to that CPU right it cannot now move anywhere.

So, now the lock is not really you know you can say that the lock is actually for the CPU and not for the thread, on the other hand if you did not disable the interrupts it is the lock should have been per thread. Because if the thread gets switched out and goes to another CPU, then that thread is holding the lock not the CPU that is hold in the lock right. But because you disable the interrupts now the thread is stuck to that CPU. So, you know you can call it a per CPU lock instead of calling at a per thread lock right but.

So, both have these are sort of interrelated, but against the point is that if you did not care about atomicity with respect to the interrupt handler you did not have to do pushcli all right. So, that is acquire right. So, let us look at the other things, so of course you know it try this is this loop we are very much familiar with just you know trying to atomically set it to 1.

And if it is not; if it is not if it does not find it to be 0 it just keeps spinning right that is a spin loop, we know this. And then it just says you know just sets a CPU variable to that I am holding this lock basically, is it to put lk dot CPU above while loop.

Student: This one not.

Firstly, until and unless you have acquired it you should not. Secondly, you know they can be racing accesses to cpu right, because this while loop is protecting. So, you can be show that only one thread or one CPU is within this region, but if you put this above it then you know there is a race condition on the cpu field of the lock right. So, because you put it after while you can be show that there is no racing and then let us look at release.

(Refer Slide Time: 09:43)

```
1500 // Release the lock.
1501 void
1502 release(struct spinlock *lk)
1503 {
1504     if(!holding(lk))
1505         panic("release");
1506
1507     lk->pcs[0] = 0;
1508     lk->cpu = 0;
1509
1510    // The xchg serializes, so that reads before release are
1511    // not reordered after it. The 1996 PentiumPro manual (Volume 3,
1512    // 7.2) says reads can be carried out speculatively and in
1513    // any order, which implies we need to serialize here.
1514    // But the 2007 Intel 64 Architecture Memory Ordering White
1515    // Paper says that Intel 64 and IA-32 will not move a load
1516    // after a store. So lock->locked = 0 would work here.
1517    // The xchg being asm volatile ensures gcc emits it after
1518    // the above assignments (and after the critical section).
1519    xchg(&lk->locked, 0);
1520
1521    vaclio;
1522 }
1523
1524 // Record the current call stack in pcs[] by following the %ebp chain.
1525 void
```

So, this is the release function and once again there is just debugging aid which is if not holding lock then you know you panic that you trying to release a lock that you not holding that is fine and you set cpu to 0 and then you exchange lock with 0. So, once

again why do I need an exchange it, so I could have just set lock dot lock 1 k dot locked is equal to 0. But instead the programmer chooses to use the exchange instruction to do this why.

Student: (Refer Time: 10:10).

Because, so that there is a fence there is a barrier between, so that this neither the hardware nor the compiler is able to reorder these instructions in anyway all right. So, that these instructions are serialized with respect to each other. In other words, in other words exchange instruction act as an implicit fence or barrier right.

So, the exchange instruction acts as the barrier, so these instructions cannot get reordered with the other instructions because of the exchange instruction. If you had just written 1 k dot locked is equal to 0, the hardware was free at run time to just reorder these instructions right as we have discussed before all right.

Now, is it ok to put 1 k dot CPU after exchange 1 lock dot 1 k dot locked, I mean is it ok to swap these two instructions two these two statements 1508 and 1519.

Student: No.

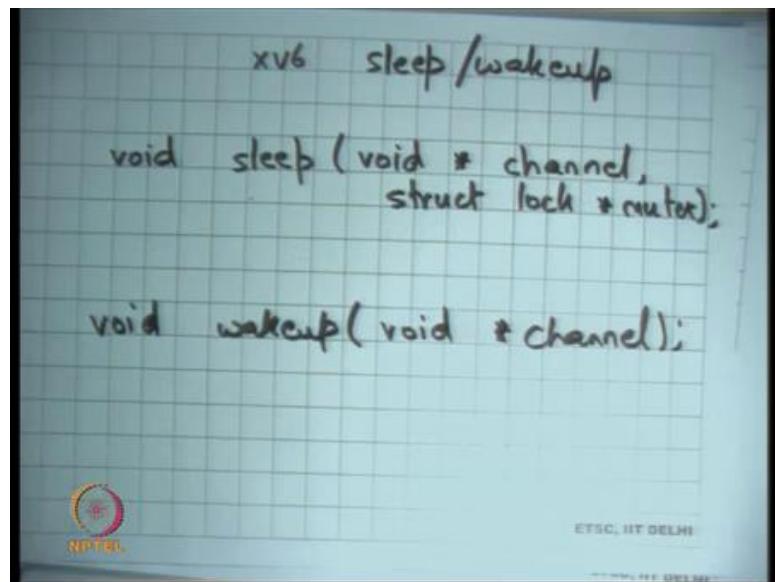
No, because there be race conditions on the statement right as soon as you release the lock other people can now.

Student: (Refer Time: 11:13).

Well other people can now access the CPU variable because, another thread could have taken the lock and now it may be trying to write to the lock. And here you are trying to write the locks they are two concurrent access to the CPU variable and so now there are two concurrent access to the CPU variable.

So, you know quite obviously and smartly I am just with the that programmer is just using the same locked field to also protect the internal structures of the lock. Internal fields of the lock variable itself apart from the critical section the fields itself have been protected by this locked field all right.

(Refer Slide Time: 11:49)



Now, let us look at sleep and wake up all right xv6 sleep and wake up this is just you can call it wait and notify. Now, these are just this is just an incarnation of condition variable and let us see how they are used. So, sleep the so the implementations are on sheet 25 of sleep and wake up sheet 25 and 26 and let us see what the idea behind sleep and wake up.

So, you have a function called sleep, let us look at the signatures of the function. So, there is a function called sleep that takes an argument void star channel, let us discuss what the channel is and struct lock *mutex all right and then there is wake up and wake up just says void star channel all right.

So, this is very similar to you know wait except that instead of a condition variable I am using this pointer called void star channel right. Instead of saying struct cv *cv I am saying void *channel. And similarly here I am saying void *channel right apart from it the semantics are exactly identical sleep is going to wait star waiting on the channel and release the lock release the mutex atomically and wake up is going to wake up in all the threads that are waiting on the channel all right so identical semantics.

Now, question is how I can just say void star channel why I do not need a struct cv here. Recall that unlike locks and semaphores the condition variables are a stateless abstraction, you do not need any state inside the condition variable you do not have any

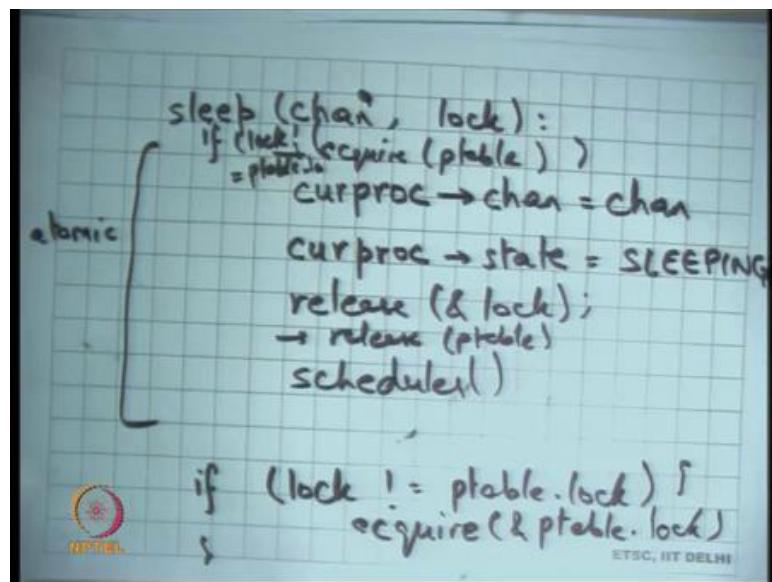
field call logged or anything that is (Refer Time: 14:00) counter or anything, so it is a completely stateless abstraction.

So, you do not even actually need to have a variable specifically, all you need to say is that there is some number that I am going to sleep on right. That number in the condition variable examples was the address of that condition variable, instead of that address of the instead of actually wasting memory you know using memory to declare that condition variable.

If you just say I am going to sleep on some channel in the sum number that number may be tied to the logic of your program, you do not need to. So, earlier I was a declaring the condition variable for the logics. So, it is example I had not full and not empty, I am saying I do not need to declare these variable because these variables do not have any state anyways.

So, why I am wasting some memory may be instead of doing that let us just use some channel which is just a number right. So, condition variable was also just being used as a number as an address. So, why do not we just use it an as an address all right this is fine.

(Refer Slide Time: 15:04)



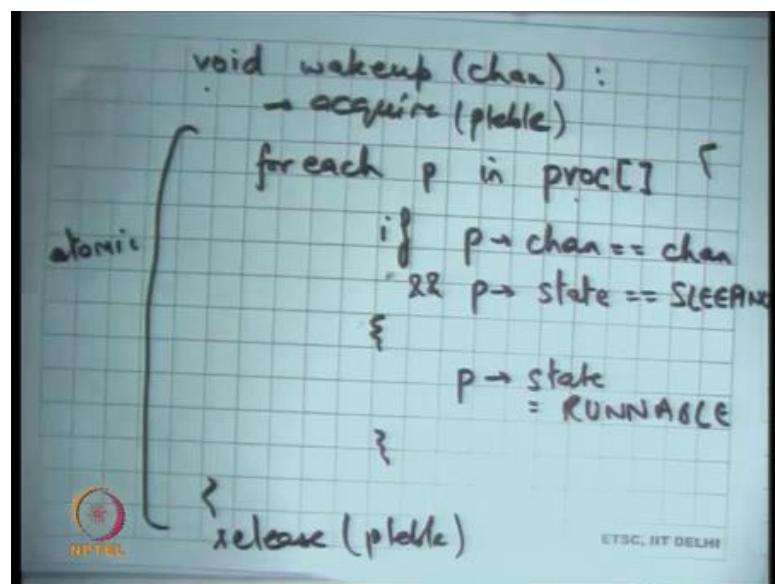
So, let us see I mean at a high level what does a sleep do sleep is going to. So, it takes two arguments I am just going to say let say whoever whichever processor process calls sleep is going to see curproc.channel = channel and curproc.state = sleeping and it is

going to release the lock right and then it will call the scheduler. So, that somebody else can get to run now.

So, he is made my it is own state as sleeping and now when it runs a schedule as the scheduler as not going to schedule this process, because this process is state has become sleeping right. So, I am just trying to develop the implementation of sleep, how it would be implemented under the covers we have discuss the semantics of condition variables already all right.

Now, let us look at wake up and this whole process has to be atomic right. What does atomic mean no other process should be able to inspect curproc while I am doing all this right. For example, if somebody calls wake up then he should not be able to you know inspect curproc while I was sort of in the middle of doing sleep right. So, how I am going to make it make it atomic let us refer the discussion but let just say this is atomic.

(Refer Slide Time: 16:36)



And then let us see how wake up is implemented, a wake up is just this it will just for each proc for each p in the proc table right. So, I have a global list of all the proc or a process if $p.\text{channel} = \text{channel}$ right and $p.\text{state} = \text{sleeping}$ then do what?

Student: (Refer Time: 17:30).

P dot state is equal to Runnable right. So, do not we change p arrow channel it will does not matter I mean channel is only valid if the state was sleeping. If the state is runnable

and there is no, I mean channel is meaningless, do I set all the processes to be runnable or only one.

Student: All.

All right that is the semantics of wake up or notify that all the processes are going to wake up, that is the semantic that we have discuss the different semantics in literature. But you know that semantics that we are working that wake up is going to wake up all the threads all right ok. Once again, this whole thing needs to be atomic.

What does it mean for this whole thing you need to be atomic? While I am doing this check no other process should be able to read or write the stuck proc structure right. This proc table should be sort of exclusive to me right or for example this one is also touching curproc is also part of the proc table right.

Student: Yes.

Curproc is just a pointer inside your proc table. So, even this should be atomic with respect to that right. So, in other words you know it should not happen then the in the middle of while I am in the middle of this code gets to run right. So, sleep and wake up need to be atomic with respect to each other. How do you ensure atomicity?

Student: By a lock.

By a lock right, so what do you think should be the easiest option here just have a lock for the entire p table.

Student: (Refer Slide Time: 18:56).

Right got enough, so just have a lock for the entire ptable have acquired ptable here and release ptable somewhere here right.

Student: (Refer Slide Time: 19:10).

Let us say acquire ptable release ptable acquire ptable by a, when I say acquire ptable I am releasing acquire ptable dot lock right and then release ptable ok. So, it makes it sort of atomic is this correct, is there a problem of deadlocks; there will be a problem of deadlocks?

Student: Deadlock (Refer Slide Time: 19:59) ptable (Refer Slide Time: 16:00).

Sure. Firstly, if the lock that we are trying to acquire is a ptable lock, then you know the same thread. So, if this argument is also equal to ptable lock then there is a problem. So firstly, I need to handle this special case because I am inside the kernel it is possible that somebody wants to sleep and the mutex that is protecting that critical section is actually the ptable lock itself.

So, I could just say if lock is not equal to ptable lock, then acquire ptable lock right. Let me write it clearly if lock is not equal to ptable dot lock acquire ptable dot lock right. Similarly, when I release the lock here. I do not need to do two releases I just need to do one release. If lock was equal to p dot ptable.lock is this am I safe now? Interrupt handlers also acquire ptable lock is that a problem?

Student: Because we have (Refer Slide Time: 21:09).

No because we know the acquire function will disable interrupt completely. So, if I am within this region, I can be show that interrupts are disabled. So, no interrupt handler will get to run, so that is the that is a whole idea by we had pushcli inside acquire right. Do not we need to worry about one more thing to ensure that there are no deadlocks.

Student: Order of acquire.

Order of acquisition of locks, do not we see that there are two locks here one is the lock that is already held before we called lock sleep, and another is ptable dot lock. So, there are two locks that are going to be acquired, assuming lock was not equal to ptable lock. So, you know the somebody the caller called acquire ptable lock then you called sleep is going to acquire ptable dot lock.

So, now you have two locks and whenever you hold there is any point you can hold two locks you have to worry about the order, otherwise they can be a deadlock right. So, s this is a; this is a real example of you know why locking kills modularity you have to worry about these things ok.

So, the convention in the operating system is that ptable lock will always be the lowest lock that will ever be taken. So, the ptable has the least priority in some sense right. So, it will be always be the it will be the always be the last lock that is ever taken. So, if you

make that assumption then this code is correct right. But that assumption has to be obeyed by all other parts of the code, that you will never acquire in another lock after you hold the ptable lock.

So, ptable lock will always be the innermost lock in other words right. So, you have to have this global and invariant across your code basically all right. Let us look at so this is you know I have sketched the implementation but let us look at the real implementation on sheet 25.

(Refer Slide Time: 22:52)

```
2551 // Reacquires lock when awakened.  
2552 void  
2553 sleep(void *chan, struct spinlock *lk)  
2554 {  
2555     if(proc == 0)  
2556         panic("sleep");  
2557  
2558     if(lk == 0)  
2559         panic("sleep without lk");  
2560  
2561     // Must acquire ptable.lock in order to  
2562     // change p->state and then call sched.  
2563     // Once we hold ptable.lock, we can be  
2564     // guaranteed that we won't miss any wakeup  
2565     // (wakeup runs with ptable.lock locked),  
2566     // so it's okay to release lk.  
2567     if(lk != &ptable.lock){  
2568         acquire(&ptable.lock);  
2569         release(lk);
```

So, this is sleep on channel and spinlock lock right that is the and once again there are there are these debugging aids, that the current process should not be 0 proc is a global variable which is indicating. What is the current process that running on the CPU it is a per CPU variable if lk is equal to 0? So, lock should not be equal to null I should not be sleeping without the lock. So, lock should not be equal to null these are just debugging aids you can ignore it.

(Refer Slide Time: 23:21)

```
2558 if(lk == 0)
2559     panic("sleep without lk");
2560
2561 // Must acquire ptable.lock in order to
2562 // change p->state and then call sched.
2563 // Once we hold ptable.lock, we can be
2564 // guaranteed that we won't miss any wakeup
2565 // (wakeup runs with ptable.lock locked),
2566 // so it's okay to release lk.
2567 if(lk != &ptable.lock){
2568     acquire(&ptable.lock);
2569     release(lk);
2570 }
2571
2572 // Go to sleep.
2573 proc->chan = chan;
2574 proc->state = SLEEPING; → release(lk)
2575 sched();
2576
```

Here is a check if lock is not equal to ptable lock acquire ptable lock and release the real lock, the lock that was used to call it ok. What is happening? So, what did I have I had something different I said if lock I said if lock is not equal to ptable lock acquired ptable lock, else do not acquired ptable lock you know you already have ptable.

Here he is saying if lock is not equal to ptable lock acquired ptable lock that is the same. But he also releasing the lock here, in my code I said you know let us I was releasing lock here does not matter if I release the lock here as supposed to release in the lock here.

So, I was releasing the lock here he is releasing the lock here does not matter does not matter, because you know you already held ptable lock. So now, you have created mutual exclusion between you know between the data. So, the data that you are going to access is mutually exclusive because, so is so this area is mutually exclusive because of ptable lock ok. Because you all you are doing is accessing the proc structure and assuming that you hold the invariant, that whenever you touch the proc structure you will hold the ptable lock this area is anyways protected from race conditions.

So, you do not really need to hold the lock for all that long you could have held it, but you know it is also to just release it a priory all right. So, let say I have released the lock here and I am somewhere here. Let say in other thread calls wake up at this point what will happen, another threads call wake up. So, wake up will try to acquire the ptable lock

and you will not get it right, it will wait for this whole thing to complete the scheduler to get called.

The scheduler recall is going to cause the ptable lock to get released whoever gets to run next is going to release the ptable lock. Recall that the schedulers invariant was that whoever calls the scheduler should have held the ptable lock and then call the scheduler and the schedulers going to schedule somebody and that somebody is going to release the lock for you

So, this was sort of very very different kind of the structure where one thread acquires the lock and another thread releases the lock right. So, this was the different pattern and so that is fine the ptable lock will be released at this point and, but you know you have ensured mutual exclusion with respect to the wakeup function.

So, if we had another acquisition of a lock here, then it would be much more prudent to release the lock as soon as possible. So, that you do not you do not have more ordering dependencies right. Because if you have another acquire here then you know you would have had to have an ordering dependency between the `lk` that you have as an argument and this particular lock also, but you know.

Student: (Refer Time: 26:06).

But you do not have any acquisition that is true all right and let us look at the wake-up function.

(Refer Slide Time: 26:14)

```
2613 void
2614 wakeup(void *chan)
2615 {
2616     acquire(&ptable.lock);
2617     wakeup1(chan);
2618     release(&ptable.lock);
2619 }
2620
2621 // Kill the process with the given pid.
2622 // Process won't exit until it returns
2623 // to user space (see trap in trap.c).
2624 int
2625 kill(int pid)
2626 {
2627     struct proc *p;
2628     acquire(&ptable.lock);
2629     for(p = ptable.proc; p < &ptable.proc[NPROC];
2630         ; p++)
```

Well here is the wake up function, the first thing it does is acquire ptable lock calls this help a function called wakeup 1 and releases the ptable lock all right and what is wakeup 1 doing let us see wakeup 1 is just above it.

(Refer Slide Time: 26:26)

```
14:35 2012 xv6/proc.c Page 10

// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)

    struct proc *p;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;

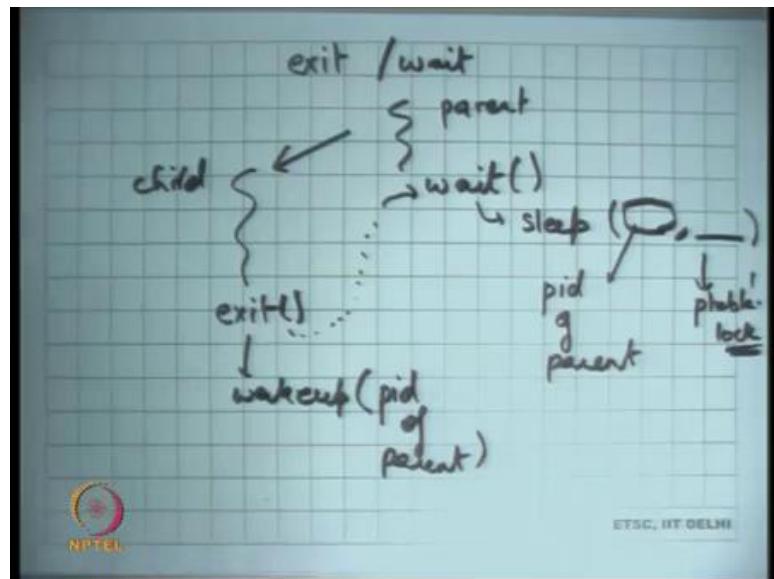
// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
```

Wake up 1 is just iterating over the ptable and checking if state is equal to sleeping and channels equal to chan then state is equal to RUNNABLE right. So, this whole logic is being called with ptable lock held all right ok. So, we understand how sleep and wakeup can be implemented under the covers in a and recall that you know.

In doing this we have make sure that the sleep is not just going to sleep putting the process to sleep in sleeping state. But also releasing the lock and is doing it in an atomic fashion in the sense that no other process can see the state of a process within this whole thing and the construct that is providing you this atomicity is what ptable lock right.

So, ptable lock is ensuring that there is atomicity in going to sleep and releasing the lock all right. So, let us look at how sleep and wakeup are used inside the xv6 kernel all right. So, we have seen before; we have seen before this the system calls called exit and wait all right. In our discussion on unique system calls and also in your lab you may have come across exit and wait.

(Refer Slide Time: 27:59)



What are the semantics of the exist and wait let us just revise that? So, let us this is the process that calls exit and here is a parent process. So, parent child so if a parent calls wait, then the wait will never return till one of it is children exits right. So, the semantics of wait is that it is going to wait till one of the children have exited. So, the dependency is that you know you will come out of it only after this as called exit right.

So, how does how will an OS implement let us say something of this sort. So, what are the semantics that I have to put myself to sleeping state till some of one of my children exits and so he will put me in the runnable state right. One way to do this is using condition variables or using sleep wake up.

So, the idea is wait internally will call sleep, it will check a condition the condition is has a you know do I have any children and if so, are they running and if so, I will start sleeping on some channel. And then exit should call wake up on that particular channel right. So, that if there was a parent sleeping on that channel then he will wake up and then he can actually come out of the wait.

Student: So, why we are using this semantics rather than child one or signals, why we use signals to wake up the parent (Refer Time: 29:32) condition variables.

No so the signal is just an abstraction at the process level right. In either case you this is I am talking about inside the kernel I am not talking about at the user level. So, inside the kernel you know let us say one process as called wait right. So, exit wait is completely independent of the sick child right. Sick child was there if the parent has not called wait and yet you want to basically inform the parent that I have you know my child has your child has basically exited.

Student: Semantics same type of thing over same process.

So, I mean Unix has both exit wait and sick child you know signal mechanism. So, I am right now talking about exit wait right. So, how is exit wait implemented, even you know signals will require some kind of synchronization it may not require sleep and wake up necessarily. We will require some kind of synchronization which will be very similar basically. But let us I mean if this is a simpler thing to worry about signals is more complicated to implement right and actually explicit does not support signals at all.

In any case I mean even Unix or Linux whatever will have exit wait without any sick child or anything right. So, if a parent is waiting then and a process exits then it is going to wake up the parent. So, that it comes out of the wait ok. So, and I am saying that you know the waiting can be implemented using sleep and telling the parent that I have existed can be implemented using wakeup. I can use what is the lock what is the mutex I should use here inside my sleep and wakeup.

If I want to do it one could mutex to use this ptable dot lock right, after all I am putting myself to sleep which basically you know I could potentially use ptable dot lock. And so let say you know that is easy I can use ptable dot lock I need any mutex that is going to

ensure that you know anything that is going to it is going to give me mutual exclusion with respect to accesses to my proc structure.

So, one lock that gives you mutual exclusion with respect to accesses to your proc structure it is basically ptable dot lock. So, that is easy.

Student: Sir, but that cannot ensure that only (Refer Time: 31:52) child can be include anybody else any process can be.

No that is just the mutex that is just for the mutex.

Student: (Refer Time: 31:58).

Right, now the question is what is the channel what is the channel I should use?

Student: Process id of parent.

Process id of parent right I can say you know I can say parents process id let say does that make sense.

Student: (Refer Time: 32:14).

Why does this channel make sense let us say I have ten children all of them are going to call wakeup on one process id of parent right? So, that is the; that is the channel that I am sleeping on right. And so, I actually make sense because pid of parent is basically you know it is assuming that your convention is that you are going to use pid of parent only for this exit wait business. Then you know another person's child cannot wake you right.

So, one a process cannot be woken by another processes child, you will only be woken up by your own children right. If you use this convention, what if you used some you know instead of using p id of parent let say I just used one constant number 1. So, I am always going to sleep on identifier on this constant called 1 let say.

Student: Any process with exit will (Refer Time: 33:13).

So, any process that exits is going to call wakeup on all the processes that are waiting is that a correctness problem?

Student: Yes.

That may not be a correctness problem because you recall that usually you would, when you come out of sleep you check the condition again. And so, what will happen is if you are sleeping on some global channel that shared across all the processes, then you know you will wake up all these other processes. But all these processes if you have written your code correctly again going to check the condition and everybody accept your own parent is going to find the condition to be false.

And so, all of them going to go back to sleep your parent is going to come out it is wasteful it is not a; it is not incorrect right. So, choosing the channel identifier is just a matter of you know figuring out.

Student: (Refer Time: 34:00).

What are you know what are my; what are my communication patterns and then figuring out the channel the identified that is common to this to the pattern? So, in this case it is a parent id. But if you something which was more course grained right like I took the extreme example of a global identifier called 1. Assuming you have enclosed your sleep loop with a while condition and a check that still correct, but it is wasteful all right. So, let us look at the exit and wait implementations the real ones on sheet 23 and 24.

(Refer Slide Time: 34:39)

```
353 void
354 exit(void)
355 {
356     struct proc *p;
357     int fd;
358
359     if(proc == initproc)
360         panic("init exiting");
361
362     // Close all open files.
363     for(fd = 0; fd < NOFILE; fd++){
364         if(proc->ofile[fd]){
365             fileclose(proc->ofile[fd]);
366             proc->ofile[fd] = 0;
367         }
368     }
369
370     put(proc->cwd);
371     proc->cwd = 0;
372     RPTMUL;
373     acquire(&ptable.lock);
```

So, that is your exit function, this function will be called if you if a process makes the existing system call and well it just says you know here some debugging aid the first

process the init proc should not get ever exit. So, that just debugging it. Close all the open files right we have seen this every process has file every table file table and so, xv6 also implements these Unix semantics and so if a process is exiting just close all the files ok. So, we are that is fine that is easy.

(Refer Slide Time: 35:14)

```
2
3 acquire(&ptable.lock);
4
5 // Parent might be sleeping in wait().
6 wakeup1(proc->parent);
7
8 // Pass abandoned children to init.
9 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
10    if(p->parent == proc){
11       p->parent = initproc;
12       if(p->state == ZOMBIE)
13          wakeup1(initproc);
14    }
15 }
16
17 // Jump into the scheduler, never to return.
18 proc->state = ZOMBIE;
19 sched();
20 panic("zombie exit");
21 } MPTBL
```

Also, you know this is some so the process that was open that was currently running, it is current working directory that cwd is also sort of in the file system knows that this process is accessing this particular directory. So, that file so nobody can actually delete that directory from under its feed. So, for that reason you know there is a way to lock that directory and we are going to discuss this when we discuss file system.

So, it is basically unlocking the directory saying that I am going to exit. So, I can unlock the directory if some other process wants to delete that directory is free to do that ok. And I said my cwd to 0 current working directory and now is my logic to do implement exit no to do exit wait synchronization.

So, I do acquire ptable.lock and I wake up on proc dot parent. So, instead of the pid of the parent, I am using the pointer of the pc. So, here I am actually using the pid itself. So, proc dot parent actually the pid of actually it is the pointed to the proc structure of the parent right.

So, instead of using the pid just use the pointer and the and the proc structure that is equally good right there is a one to one correspond pid in the proc structure. So, proc dot parent you just wake it up right, notice that I am not calling wakeup I am calling wakeup one because I have already the ptable lock right.

So, here is another example why locks skill modularity, I have to worry about which function will acquire the lock and which will not acquire the lock what locks I already have you know. So, it is not nice ok. So, I will just wake up the proc dot parent and just fall through and if I have any children then, so then I just iterate over the proc table notice that I have the ptable lock. So, I can safely assume that these accesses are atomic.

So, ptable.proc to p so look at all the processes, if I am the parent of that process right which means that this process is my child, then do what I am going to you know I am going to exit. So, basically, he is you know the offend process has to be connected to the init process, recall that this is something this sort of semantics that we discuss.

That if the parent exits then all its children are connected to the init proc and init proc is going to call wait. So, that there are no zombies remaining right. And if p dot state is equal to zombie then wake up init proc what is happening here. So, init proc is probably already call is already in the wait loop right. So, init proc is just constantly calling wait, just to collect all the sort of zombie processes and now.

So now, I was the process and I had a child process I forgot to call wait on the child process. So, that child processes just languishing as zombie there right. Now I am going to I am exiting. So, what I can do is all my zombie process if I see any zombie process, I can tell the init proc to wake up, because I have attached a zombie process to him.

So, now his wait you know wait is going to return, so that is what I am doing. So, I am waking up the init proc, so that his wait returns if he is inside the wait right. This is to wait to ensure that your parent returns. So, a process that is exiting you are making sure that your parent returns from a wait, this is to make sure that init proc returns from the wait if any if it is inside right, because you just attaches zombie process to the init proc is right.

And then you make your state as zombie right, notice that I am not cleaned up my virtual memory, I will not cleaned up my case stack I will not de-allocated my virtual memory

my page table still remains all the data that I have I am not freed it right. I have just mark my state as zombie and call the scheduler what is going to happen is that the my parent process when it is going to run is going to call wait and that is going to de allocate my page table and my case stack ok. Why cannot I just de allocate my page table here.

Student: (Refer Time: 9:22).

Because I am Standing on that page table right, that is the page table that is currently loaded. This exit function is being called in the context of the process that is currently running.

Student: Yeah.

And so, I cannot just de allocate the page table because that page table is actually the one that is loaded into the hardware right. Now you only change your state to zombie and now when the other process gets loaded now the you know now you are off the CPU and now you can be de allocated.

Similarly, case stack cannot be de allocated because this entire function is executing of that case stack. You cannot just de allocate the case stack it is your parent who will de allocate this case stack (Refer Time: 39:58).

Student: Sir, but we can de allocate the user page table as an.

You can de allocate the user side of the page table that is an optimization and you know the real operating system will perhaps do that. But you know xv6 just makes it simpler and says the parent will de allocate everything ok.

Student: Sir in case of (Refer Time: 40:19) these scheduler functions was de allocating the case stack as an. So, is that also possible optimization as in the case stack once we will reach the scheduler function and you have seen that the process from which you have just come into that has not exited. So, what I do is it de allocate.

Right, so the question is instead of waiting for the parent to de allocate the stack, is it possible to just you know on every context switch check if my previous process has become zombie. And if you know on the context switch as soon as your context switch.

Student: (Refer Time: 40:52).

So, you have figured out that the previous process that just got context switched out as become zombie then you can just de allocate it is structures. But you know only some of it is structures you still have to maintain it is proc structure, because the parent may call wait. You have to for example, store the exit status return value exit code right.

So those are all possibilities, but you know that there is a tradeoff in doing that. On every context which you are going to check whether the previous process context which process has become zombie or not right that itself has a cost some cost to it. So, you know whether it is a worth file thing to do or not i is something that has a that is (Refer Time: 41:27) basically a design decision ok.

(Refer Slide Time: 41:33)

```
1 struct proc *p;
2 int havekids, pid;
3
4 acquire(&ptable.lock);
5 for(;;){
6     // Scan through table looking for zombie children.
7     havekids = 0;
8     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
9         if(p->parent != proc)
10             continue;
11         havekids = 1;
12         if(p->state == ZOMBIE){
13             // Found one.
14             pid = p->pid;
15             kfree(p->kstack);
16             p->kstack = 0;
17             freevm(p->pgdir);
18             p->state = UNUSED;
19             p->pid = 0;
20             p->parent = 0;
21             p->name[0] = 0;
22             p->killed = 0;
23     }
24 }
```

Let us look at wait, so here is wait I just go through my so. Firstly, I acquire the ptable lock, once again what is wait going to do it is going to go through the entire ptable and it is going to check if I am the parent of this particular process and if it is zombie, then I can just return right I called wait on something that is already existed. So, I can just return right because that is already zombie. So, I can just return from there.

So, I just sort of go through the ptable, if I am not the parent of that proc of that p then continue. Otherwise I am the parent I just check if it is a zombie. I found the zombie my wait does not need to sleep my wait can just return. So, I found one I can just free all it is things. So, I can free it is case stag, I can free it is page directory, I can set it is state to unused right.

(Refer Slide Time: 42:18)

```
if(p->parent != proc)
    continue;
havekids = 1;
if(p->state == ZOMBIE){
    // Found one.
    pid = p->pid;
    kfree(p->kstack);
    p->kstack = 0;
    freevm(p->pgdir);
    p->state = UNUSED;
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->killed = 0;
    release(&ptable.lock);
    return pid;
}
// No point waiting if we don't have any children.
if(!havekids || proc->killed){
    release(&ptable.lock);
```

And I am doing this under the protection of ptable lock recall and I just release the ptable lock and I return the process id of the process that I just released right that the that is the one that was zombie. So, return the pid.

Student: Sir do we need to actually put set these field to 0 because anyways once you have drafted and use, we are not going to access (Refer Time: 42:42)

Sure. I mean you do you need to set it to 0, because you have set it to unused, I mean those small things can be done right. I am not sure you know what other parts of the code are relying on you know how what invariants they maintain but you could do that sure all right.

(Refer Slide Time: 42:55)

```
2426     p->killed = 0;
2427     release(&ptable.lock);
2428     return pid;
2429   }
2430 }
2431
2432 // No point waiting if we don't have any children.
2433 if(!havekids || proc->killed){
2434   release(&ptable.lock);
2435   return -1;
2436 }
2437
2438 // Wait for children to exit. (See wakeup call in p
2439 sleep(proc, &ptable.lock);
2440 }
2441 }
2442
2443
2444
2445
2446
```

Now if I do not have kids right which means I do not have any children and I called wait I do not need to wait for anything right because I do not have any children. So, and I am calling wait. So, I just return with a minus 1 status that is all right.

And then there is a field called proc dot killed I am going to discuss this next time let us wait on that. But at this point I know that I have some children this process has some children and I have some children that are not yet zombie. So, what I have to do I have to wait for them to become zombie. So, I just sleep on proc and this is the ptable dot lock that is protecting that right. So, that is basically exit wait.

So, now this process has become will go into a sleeping state as soon as somebody exits, he is going to call wake up and the channel is going to be his parent. I slept on my own id right he is going to wake up on his parents id. So, that is how this synchronizing happening ok.

Let us stop here and I am going to discuss the kill system call and how the killing system call is implemented using these mechanisms and how the id device driver. However, disk device driver uses synchronization and how device driver works internally to do this thing. And the also we are going to discuss virtual memory and we are going to start the topic of virtual memory and paging in the next lecture.

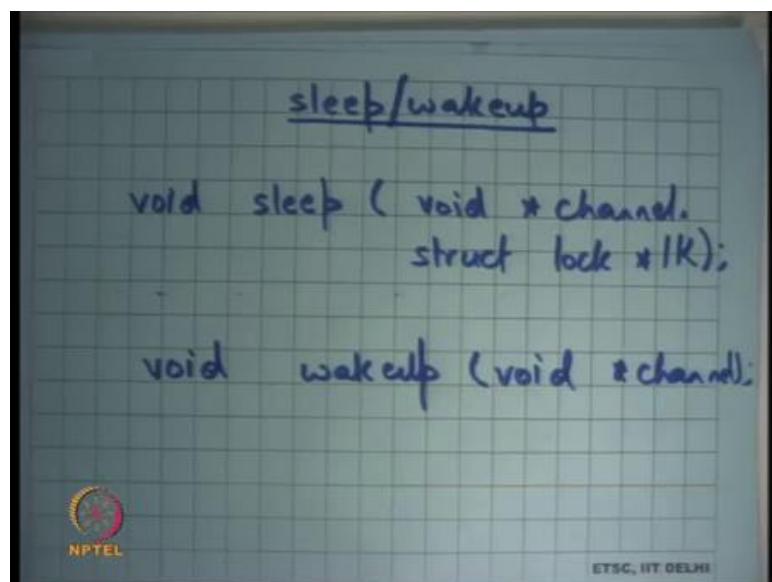
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 28

More synchronization in xv6: kill, IDE device driver; Introduction to Demand Paging

Welcome to Operating System lecture 28 right.

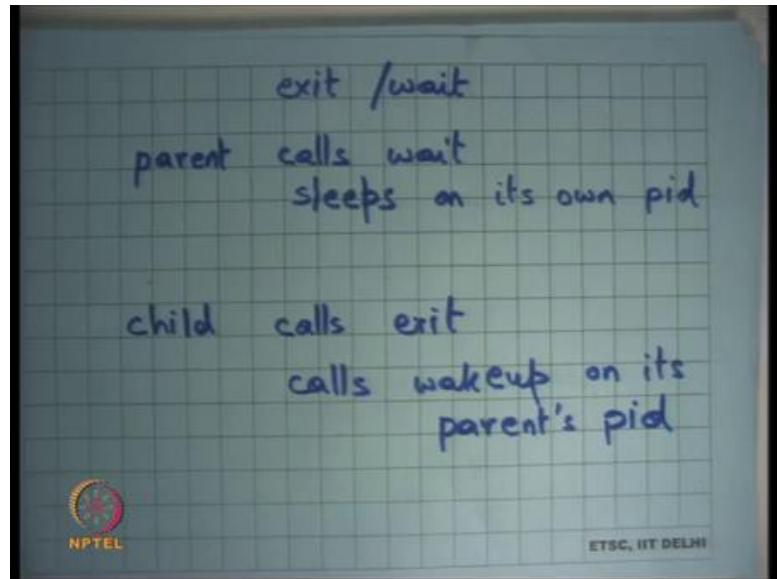
(Refer Slide Time: 00:29)



Last time we were discussing sleep wakeup on xv6 and we said sleep wakeup is very is actually identical to condition variables except that we do not explicitly declare a condition variable we just sleep on some channel and that channel is some identifier. And, earlier in condition variable you would have declared a condition variable associated with that channel.

Now, you do not need to declare anything separately. So, that way you do not have because a condition variable has no state you do not need any space for the condition variable. So, really this channel is nothing, but an integer right any 32 bit integer that can be used to indicate that I am sleeping on this channel and then you use the same integer to indicate that I want to wake up all sleepers on this particular integer right. So, that is what it means.

(Refer Slide Time: 01:16)



And we said that you know we looked at some examples we said look there is exit and wait you know we understand this exit and wait system call in Unix and x v 6 implements the same. So, parent calls wait let say and if it finds the child that is currently not a zombie and running then or active then it is sleeps on it is own pid right. So, the channel is basically it is own pid and if a child calls exit thereafter then it calls wake up on it is parents pid. And, so that is how you basically ensure that you know you only wake up your own parents and nobody else.

And, we also said you know instead of this convention if we if I had a more general convention where instead of sleeping on my own pid I sleep on some global identifier that would have been also correct provided you are enclosing the wait enclosing the sleep inside in a loop which is checking the condition right. So, if a child calls wakeup on all the parents then all the parents are going all the waiting parents are going to come out of the sleep only to go back to sleep alright.

So, that is it is wasteful. So, it is better to have as fine-grained channels as possible and so this is a, this makes lot of sense. It is exactly as fine-grained as you can get in this particular example right.

Student: Sir.

Yes.

Student: Sir in a current case a when you are sleeping on your own pid and child is waking up the parents pid to be needed to check for the condition because at most only one process can be woken up by a child

Ok.

Student: So.

Alright. So, the question is you know in this case if child you know if I am using this convention that parent is waiting on a it is own pid and child is calling wake up on it is parents pid do I need to enclose the sleep in a while loop, can I just you know use if you know. So, I can be sure that if a parent has woken up from sleep then definitely one of it is children is zombie and so it can be collected right.

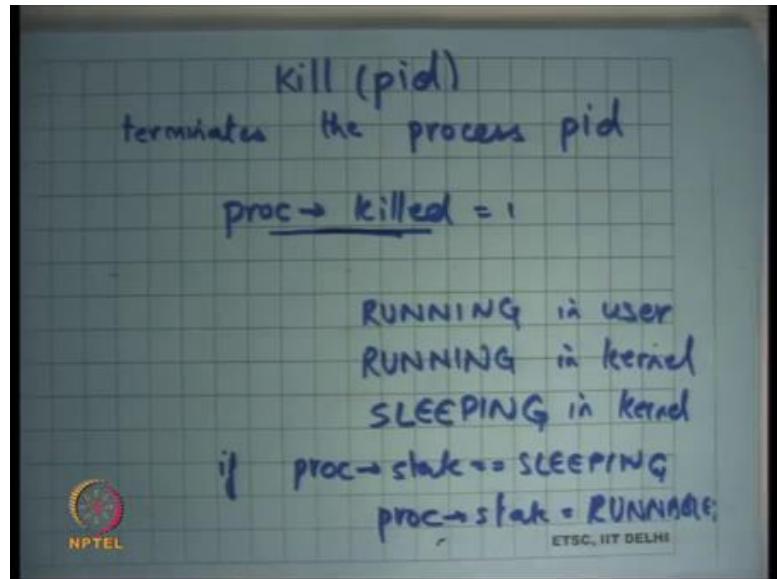
Well, I mean yes you can do such reasoning that is we have seen such a thing in single producer single consumer case, but you know doing these kinds of reasoning is usually error prone. You know in an as I am going to show you next there are other checks that you do make whether the process has been killed in the middle or things like that.

So, it is always better to you know put it in you know recheck after coming out of a wait that whether the condition is true or not and the overhead of rechecking is very small anyways right it just memory operation to basically do it also it is possible that you know. So, you basically said I want to I am going to use this convention for exit wait.

But let say completely unrelated part of the code wanted to use some other synchronization and it also decides to sleep on the pid of some process right. So, let us say there is some synchronization is going on due to some other completely unrelated part from exit wait exit right and so that also sleeps on pid.

And, so now they can sort of you know cross communicate and to protect from errors because of this cross communication you should always enclose your sleep within a while right. If you do not then you have to be very careful you know it become a global invariant then nobody else should be sleeping on the pid for example, right alright. So, today I am going to discuss another system call called kill right.

(Refer Slide Time: 04:36)



So, we have seen the kill system call in the context of Unix where we said that you can say kill on a pid and that will send a signal to that particular pid to that particular process. So, you can say kill pid with some signal and that signal (Refer time: 04:48) x v6 does not implement signals.

So, kill(pid) does not send a signal to the pid, but just terminates that particular process. So, idea is if I send kill on a process id then that process gets terminated. So, how you think kill can be implemented? Can I just acquire the p table lock, go to its process, and free all its structures does that make sense at all? Does that make sense? Yes.

Student: No.

No. Why not? One process called kill pid that that process just acquires a p table lock goes through the proc table finds the process for with that pid and freeze all its structures is that ok.

Student: (Refer time: 05:38).

What are those processes currently running? Right. It is in the middle of something and now you have just freed up deposit structures let say you know you freed up it is case stack and it was running in kernel mode you know suddenly there will be some bad

things suddenly happen. It could be in the middle of some operation you know it has not finish it is operation and now you free it or you completely terminate it you know.

Firstly, you know it is running. So, how can you just free it? If you free it then you have not told it to stop you have to you know some have some way of asking you to stop also it could be in the middle of something and you want it to come out of wherever it is and then exit gracefully right. And, so the way you would implement this kind of a thing is that you know you will basically have a flag inside the proc structure called killed right.

So, there is a structure called killed inside a proc and if one process calls kill pid then I will just iterate over the procs table find the process and set it is killed bit to 1 right. So, by default it is 0, but I will just set it to 1. Now, it is that process which will basically you know if it was in the middle of something then it will just you know when it comes out of it, it is that processes responsibility to know that it I have been killed and so it should exit alright.

So, if a process has been killed it should exit and exit will not free up it is data structures either it will be it is parent who will call wait and then that will free it free it up. In any case so, the point is that you know kill cannot be done immediately because the process may be running. So, the way it is run as you just set a bit in that processes structure and let that process run when the process you know finishes comes out of wherever it was, then it should really check whether I have been killed or not and then it will exit alright.

So, when you set proc dot killed you are accessing the shared proc structure. So, you should be holding the p table lock alright. So, you just hold the you use the p table lock to basically set this bit, but after that it is stack processes responsibility to basically exit after it has checked that it (Refer time: 07:53) alright.

Student: Sir.

Yes.

Student: Is any process allowed to kill any other process?

Is any process allowed to kill any other process well; if the in Unix if the processes belong to the same user then any you know a user is allowed to kill it is own process basically, but one user is not allowed to kill any another users process. Also, you know

modern operating systems like Linux have concept over root user which we all know that root user is super user. So, root user can kill any other users process for example.

So, you can have this access control mechanisms, but you know in x v 6 you just have one user; so, any process can kill any other process alright. So, I am saying that one process just sets the killed bit of the other process and I now expect that process to basically exit right. So, you know what if it does not exit, you know where what if some malicious process, how do I make sure that the that process will definitely exit in some bounded amount of time?

Student: (Refer time: 08:54) called.

So firstly, you do not trust the processes user space, but you definitely trust the processes kernel space right. Anytime the process is executing in the kernel mode it is your code that is running right, it is not the users codes that running it is your code that is running. So, you always trust the processes kernel space right because process in the kernel space is just a kernel thread and you complete trust that code. So, anytime the processes executing in the kernel you can be sure and because you have it in the kernel code you can be sure that it will call exit whenever kill bit is set.

Now, you can say that what if the process remains in the user mode and never comes in the kernel mode. But that is not going to happen because you know you have a timer interrupt that may make sure that every process periodically comes in the kernel mode.

So, whenever it comes in the kernel mode next it is going to check whether something you know whether it needs to exit or not. So, once again this orchestration has been done by the kernel programmer alright. Also let us say I said the killed bit of a process and it was sleeping when it was killed right.

So, let us take a few cases let say this process was running when it was killed and let say it was running in user space when it was killed. No problem. Eventually it will come to the kernel space either because it made a system call or because of the timer interrupt and as soon as it comes into the kernel space it should check whether I have been killed or not. And, if it has been killed you know right at the entry you should put a check that if it a have been killed then I should exit and that is a very safe place to exit

because you are not in the middle of any kernel operation; you have just come from user to kernel and you can just ask him to exit right there.

So, this is easy let say I was running in kernel mode alright. So, I am running in kernel mode, I could be in the middle of doing something of course, I am using the k stack and all that and so basically I should wait for the kernel to finish whatever it is doing and then come back quite likely. If it is running in kernel mode either it will return back to the user mode eventually. So, that may be nice place to check the killed flag. So, right just before reentering the user space you can check the killed flag and call exit before it reenters the user space right that is one thing.

If it is running in kernel mode, but it is not going to reenter the user space it is possible that you know the only other possibility is that it is going to sleep right or it is going to do something else. So, all these places where it could be going to sleep for example, before you do that, you should check that it is killed and you should check it all these sort of boundaries where you are not in the middle of something. So, at any place where you have showed that all the kernel data structures are consistent at that point you can pull put this check that if I have been killed then I should exit alright.

Now, let us say I was sleeping alright. So, sleeping of course, can only happen in kernel mode right. It does not make sense that a user is a process is sleeping in kernel mode in user mode you know. If it is in the user mode it must be running something it comes to if it wants to sleep it will come to the kernel mode and then it will call the sleep function to basically sleep.

So, let us say it was sleeping in the kernel and you basically set proc dot killed is equal to 0 what will happen is equal to 1. What will happen? The process was sleeping, it will never get scheduled and so you know it will not get you know. So, killed is not going to have any effect, it just going to sleep forever may be. So, not just do not only do you set proc dot killed is equal to 1 you also say if it was sleeping then mark it as runnable alright.

So, here is that here is something sort of dangerous looking. I am saying any process who is sleeping if it has been killed just mark it runnable alright irrespective of you know what was the condition you know on which it was sleeping what is the channel it was

sleeping on does not matter just mark it as runnable it will get scheduled eventually and when it gets scheduled it will get to run ok.

Now, it is the responsibility of the programmer to ensure that whenever a process comes out of the sleep it also checks whether the killed flag has been set or not alright and if so then it should exit ok. So, once again it is a kernel programmer who is making sure that you know. Firstly, you need to do this because otherwise you know a sleeping process will never get killed.

And if you need to do this you are violating some you may be violating some invariance because after all the process went to sleep on a some condition and that condition has not yet become true, but you are still made it runnable and so it will come back and it will basically, but before you do start doing anything you should probably check whether it has been killed and if so it should exit alright.

Student: Sir, but if it is sleeping, we can assume that after some time, it may wakeup and then we can exit.

If it is sleeping I can assume that after sometime it may wakeup and then it can exit well I mean so, let us let me think of an example where it may be sleeping and it may not wake up. So, well let see. So, let say I was a parent and I wanted to wait on my child to exit right and so my child is going to run for a long time and somebody wants to kill me right. So, should I wait for the child to you know finish its execution before the parent gets killed maybe not right.

So, you want to basically kill it right then you know there should be some bound on the time it takes between the killed (Refer time: 14:54) between the kill operation and the actual act of getting killed right. Otherwise you know you have killed it, but it still appears on your process table that is not a good idea alright.

So, and it may not be always necessary that when you come out of sleep you always check the killed and you exit you know it really depends on the semantics under which you are sleeping and I am going to sleep you couple of examples to show when you need to exit on being killed and when you do not need to exit on being killed alright. So, let just look at the code. So, this is sheet 31 and this is the kill function alright.

(Refer Slide Time: 15:52)

```
2620:
2621 // Kill the process with the given pid.
2622 // Process won't exit until it returns
2623 // to user space (see trap in trap.c).
2624 int
2625 kill(int pid)
2626 {
2627     struct proc *p;
2628
2629     acquire(&ptable.lock);
2630     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2631         if(p->pid == pid){
2632             p->killed = 1;
2633             // Wake process from sleep if necessary.
2634             if(p->state == SLEEPING)
2635                 p->state = RUNNABLE;
2636             release(&ptable.lock);
2637             return 0;
2638         }
2639     }
2640     release(&ptable.lock);
2641     return -1;
2642 }
2643
2644
```

So, the first thing you do is acquire p table lock right then you check you iterate over the p table find a process whose pid is equal to the argument, set it is killed value to 1, if it is sleeping mark it runnable release the p table lock and return alright that is it. So, now let us see where the killed is being used. So, this is sheet 26 sorry not 31 sheet 26 alright.

(Refer Slide Time: 16:22)

```
3101 trap(struct trapframe *tf)
3102 {
3103     if(tf->trapno == T_SYSCALL){
3104         if(proc->killed)
3105             exit();
3106         proc->tf = tf;
3107         syscall();
3108         if(proc->killed)
3109             exit();
3110         return;
3111     }
3112
3113     switch(tf->trapno){
3114     case T_IRQ0 + IRQ_TIMER:
3115         if(cpu->id == 0){
3116             acquire(&tickslock);
3117             ticks++;
3118             wakeup(&ticks);
3119             release(&tickslock);
3120         }
3121         lapiceoi();
3122         break;
3123     case T_IRQ0 + IRQ_IDE:
3124         ideintr();
```

Let us look at sheet 31 where the killed flag is being checked. So, this is the trap function. Recall that the trap function gets called on every trap. A trap is both a system call or an external interrupt like a timer interrupt or any exception like a page fault right.

So, all these are basically traps and so the trap function gets called. Recall that the all traps function all traps assembly code used to call the trap function.

So, trap function gets called every time and you can see here that at entry you are checking if proc dot killed exit right. Similarly, after that you basically execute the system call and on exit you are saying if proc dot killed then exit right. And, so you will see similar sort of peppering of this code if proc dot killed exit at other places in the code for example, here is another example.

(Refer Slide Time: 17:13)

```
3171 // Force process to give up CPU on clock tick.  
3172 // If interrupts were on while locks held, would need to check  
3173 if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_T  
3174     yield();  
3175  
3176 // Check if the process has been killed since we yielded  
3177 if(proc && proc->killed && (tf->cs&3) == DPL_USER)  
3178     exit();  
3179 }  
3180  
3181  
3182  
3183  
3184  
3185  
3186  
3187  
3188  
3189  
3190  
3191  
3192  
3INTEL  
3194
```

Let say if proc and proc dot killed, and I was executing in user mode then exit right. So, for example, this is the timer interrupt while I was executing in the user mode and the process has been killed since in the past then exit.

(Refer Slide Time: 17:29)

```
3154         /* Unexpected trap #d from cpu #d eip #x (cr2=0x0) */
3155         tf->trapno, cpu->id, tf->eip, rcr2());
3156     panic("trap");
3157     // In user space, assume process misbehaved.
3158     sprintf("pid %d %s: trap %d err %d on cpu %d "
3159             "eip 0x%x addr 0x%x—kill proc\n",
3160             proc->pid, proc->name, tf->trapno, tf->err, cpu->i-
3161             rcr2());
3162     proc->killed = 1;
3163 }
3164
3165 // Force process exit if it has been killed and is in user sp-
3166 // (If it is still executing in the kernel, let it keep runn-
3167 // until it gets to the regular system call return.)
3168 if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3169     exit();
3170
3171 // Force process to give up CPU on clock tick.
3172 // If interrupts were on while locks held, would need to chec-
3173 if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ
3174 yield();
3175
3176 // Check if the process has been killed since we yielded
3177 if(proc && proc->killed && (tf->cs&3) == DPL_USER)
```

Similarly, you know you will see something somewhere here right. So, so these are the places notice that he is basically identified some safe places where you should exit you should check and exit. Also, he has make sure that there is a bounded amount of time within which it will exit alright. Now, let us look at the wait and the exit code that we were looking at last time alright just to complete the discussion.

(Refer Slide Time: 18:12)

```
2401 // Return -1 if this process has no children.
2402 int
2403 wait(void)
2404 {
2405     struct proc *p;
2406     int havekids, pid;
2407
2408     acquire(&ptable.lock);
2409     for(;;){
2410         // Scan through table looking for zombie children.
2411         havekids = 0;
2412         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2413             if(p->parent != proc)
2414                 continue;
2415             havekids = 1;
2416             if(p->state == ZOMBIE){
2417                 // Found one.
2418                 pid = p->pid;
2419                 kfree(p->kstack);
2420                 p->kstack = 0;
2421                 freevm(p->pgdir);
2422                 p->state = UNUSED;
2423                 p->pid = 0;
2424                 p->parent = 0;
2425                 p->namefd[0] = 0;
```

So, here is let say the wait code right. This is sheet 24. I am looking at the wait code and recall that I was acquiring the p table lock iterating over the p table.

(Refer Slide Time: 18:26).

```
2417      // Found one.
2418      pid = p->pid;
2419      kfree(p->kstack);
2420      p->kstack = 0;
2421      freevm(p->pgdir);
2422      p->state = UNUSED;
2423      p->pid = 0;
2424      p->parent = 0;
2425      p->name[0] = 0;
2426      p->killed = 0;
2427      release(&pstable.lock);
2428      return pid;
2429  }
2430 }
2431
2432 // No point waiting if we don't have any children.
2433 if(!havekids || proc->killed){
2434     release(&pstable.lock);
2435     return -1;
2436 }
2437 // Wait for children to exit. (See wakeup1 call in proc_exit)
2438 sleep(proc, &pstable.lock);
2439
2440 }
```

And, you know and if I find that there was some child if I have a, if this process has the child that has not that has not yet a zombie then I will go to sleep right. So, let say I was sleeping. So, let say this process was sleeping here and it is really waiting for one of its children to exit.

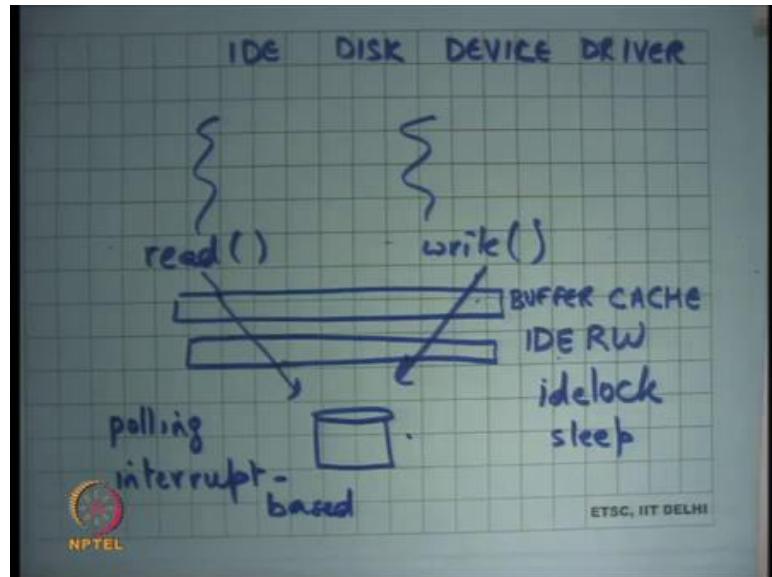
Now, somebody calls kill on this particular pid. So, what will happen it will be marked runnable it will come out of sleep immediately. Even though none of its children have actually become zombie, yet it will come out of the sleep it will be nice thing now is I will go back and check the condition again right. So, here is an example where you know you actually even though the you have chosen the channel correctly you had to check the condition again because you there are other reasons why could have been woken up right.

So, you check the condition again and once again you find that you know there is nothing that as yet become a zombie there is none of my children have become a zombie I come here. But, this time I find that I have been killed and so I will release the p table lock and return minus 1 and return minus 1 will you know eventually go back to system call exit and there you are going to check whether it has been killed then it will it is going to call actually exist alright.

So, in this loop of checking here is an example where you basically you know if somebody has been woken up from a sleeping state unceremoniously not correctly then

you will basically the programmer is handling correctly and this is very important to handle it correctly alright ok. Now, let us look at another example of how wait and notify is used and I am going to look at the IDE device driver the disk device driver.

(Refer Slide Time: 20:09)



So, IDE disk driver device driver. As you all know a computer system is made up of a CPU, main memory and a lot of devices right and one of these devices is the disk hard disk magnetic disk let say and IDE is just one interface one standard interface to communicate with the disk. So, it is a standard the IDE device manufacturer who confirm to that standard and the operating system developer will also confirm to the standard and so they can talk to each other alright.

So, let us see what is what happens basically there are multiple processes right and they may be calling read or write system calls and eventually you know the file descriptors of these read and write system calls maybe pointing to the disk alright. So, when they make a system call, they become kernel threads and these kernel threads will try to access the disk and read or write data from the disk.

So, what sets in the middle is this IDE driver alright on the xv 6 code this function is called IDE RW IDE read write alright. The multiple threads who are coming inside this trying to access the disk and they all go through this device driver to be able to access the disk. Also, because the disk is really slow as we as we already know what we also keep

here is what is called a cache right. So, if you read something from the disk you basically store it keep it in that cache.

So, that you know if some if somebody else wants to access it then it just gets satisfied from the cache and notice that this cache is the shared cache. It is shared across all the processes right. So, if one process tries to read something then it comes into the cache and another process tries to read the same thing then it does not need the disk access you save disk access because of a shared cache and so because it has to be a shared cache this cache has to be implemented inside the kernel right. If it was not a shared cache you could have implemented the cache at users space also I mean does not matter right, but because it is a shared cache it has to be a implement into the kernel and almost every kernel has it and this is call the buffer cache that is a common name for it alright.

So, buffer cache has many buffers and so it just reads data from the desk into the buffer and all subsequent request are checked against the buffer cache if you can satisfied from the cache then you just return it from the buffer cache otherwise you go to IDE RW to reach the disk alright. So, firstly there is just one disk and there are multiple logical processes right and multiple threads the threads could be logical threads or you know they could be logical concurrency or physical concurrency in either case you need to provide mutual exclusion on your access to disk. So, only one thread should be accessing the disk at any time right.

So, so that is needed. So, what basically what you do inside IDE RW you have some lock. So, on x v 6 you have an idelock. So, every thread before it acts to access the disk must acquire the lock and then access the (Refer time: 23:43). So, that make sure that only one thread is actually controlling the disk at anytime. So, one thread comes in talks to release gets it is data or writes the data reads or writes the data goes out then another thread comes in and so on. So, it is completely sequential in that sense also as we know that a disk is a really slow right.

So, if there are so, it is quite likely that there are you know lots of threads that are waiting for the disk to be available. It is this has a very slow thing then a queue get will usually get built up in front of that slow resource right and so the best thing to do would be to sleep as opposed to spin right because you the disk is basically milliseconds and you do not want to spin for milliseconds long. So, you would probably want to sleep. So,

you will basically sleep on something and so when the disk actually gets done then you will wake up the corresponding process alright.

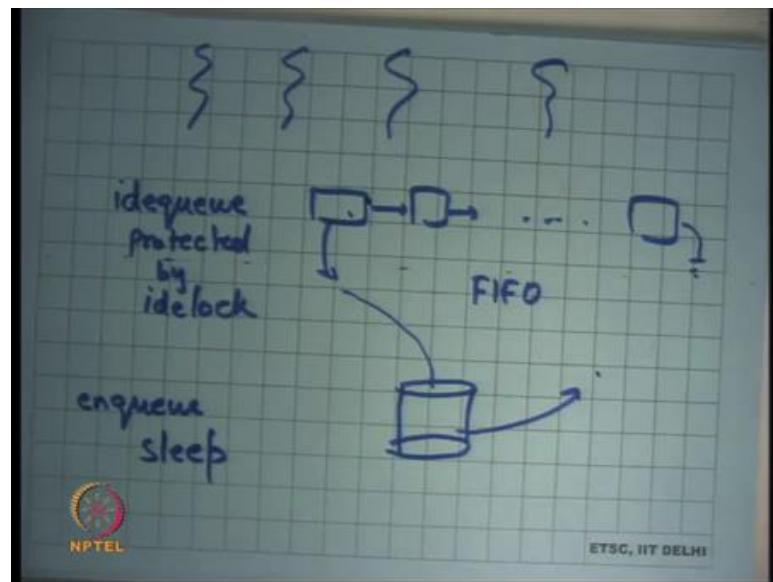
Also, device drivers are basically return in one or the two ways: one is called polling and the other is called interrupt based right. So, what this means is let say I request the disk to do something let say I ask the disk to say to read some data I want to read the sector number 10. Now, I can keep asking the disk. So, the disk is going to take some time and so I can keep asking the disk are you ready are you ready are you ready alright. So, that is called polling right you keep polling the disk for the data alright.

So, you basically just keep checking at some periodic interval whatever you think is valid you just keep polling the disk at periodic intervals and as soon as you get the data you give it back alright. Polling is alright. So, that is one way to do it the other way to do it is interrupt base where you ask the disk to do something and then you go to sleep alright and now the disk has been configured to generate an interrupt whenever it finishes.

So, I do not need to continuously keep asking it will call me back using an interrupt. So, it will generate the interrupt and the interrupt handler I when the interrupt gets generated. I can check whether my job has been completed and if so, I will do it take it right. So, there are two ways to deal with devices in general polling and interrupt ways slow devices are better dealt with in interrupt ways manner fastest devices are better dealt with in polling ways manner alright.

So, first so, the other thing is this IDE device has to be interrupt based alright. So, what is going to happen is that a thread is going to take the lock it is going to enqueue its request into a list into a queue. And, then it is going to go to sleep whenever the device finishes it is going to generate an interrupt and the interrupt handler is going to call wake up right. So, that process can actually continue. So, who whosever job is been done that process will be called that you will basically wake up that particular process alright

(Refer Slide Time: 27:09)



So, the way it works on xv6 is basically that you have a queue which it calls idequeue and this has all the buffers data waiting for disk access each and so there are multiple threads that try to add to this idequeue if you basically take. So, protected by idelock; so, if the multiple threads who want to access the disk, they actually trying to they first contained on idequeue. So, they contained an idequeue and enqueue their request on this idequeue and this idequeue is processed in FIFO order by the disk right.

Now, the disk; so, now the driver basically picks up one request from here and asks the disk to do it the disk generates an interrupt when it is done and it basically. So, whoever was the process that is that was requested this particular thing that particular process is woken up and that process can now go on its way.

So, the process enqueues its request on the idequeue and goes to sleep alright. So, enqueue and sleep and when the device takes one request at a time and when it is done then it wakes up the sleeping corresponding process alright. So, let us look at this code this is IDE RW on sheet 39 ok.

(Refer Slide Time: 28:47)

```
    nc buf with disk.  
    B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.  
    se if B_VALID is not set, read buf from disk, set B_VALID.  
  
(struct buf *b)  
  
uct buf **pp;  
  
!(b->flags & B_BUSY)  
anic("iderw: buf not busy");  
(b->flags & (B_VALID|B_DIRTY)) == B_VALID)  
anic("iderw: nothing to do");  
p->dev != 0 && !havedisk1)  
anic("iderw: ide disk 1 not present");  
  
ire(&ideLock);  
  
ppen b to idequeue.  
next NPTEL;  
pp=&idequeue; *pp; pp=&(*pp)->qnext)
```

So, this is the function IDE RW. It takes an argument buffer which is you know some point into your buffer cache and semantics are that if the buffer is you know buffer the struct buf has the field called flags and if the B_DIRTY field is set in the flags then you should write treated as a write request.

So, you want to write this buffer to the disk else if B_VALID is not set then read the buffer. So, you know whether it is a right request or a read request is encoded within the buffer in it is flags whether if it is valid if it is not valid then it is a read request if it is dirty then it is a write request alright.

(Refer Slide Time: 29:31)

```
// Sync buf with disk.  
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.  
// Else if B_VALID is not set, read buf from disk, set B_VALID.  
void  
iderw(struct buf *b)  
{  
    struct buf **pp;  
  
    if(!(b->flags & B_BUSY))  
        panic("iderw: buf not busy");  
    if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)  
        panic("iderw: nothing to do");  
    if(b->dev != 0 && !havedisk1)  
        panic("iderw: ide disk 1 not present");  
  
    acquire(&idelock);  
  
    // Append b to idequeue.  
    b->qnext = 0;  
    for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  
    ;
```

So, here are some debugging aids you know firstly alright. So, we can ignore this for a minute, and we say either it should be valid, or it should be dirty right cannot be that it is neither.

(Refer Slide Time: 29:47)

```
3959     panic("iderw: buf not busy");  
3960     if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)  
3961         panic("iderw: nothing to do");  
3962     if(b->dev != 0 && !havedisk1)  
3963         panic("iderw: ide disk 1 not present");  
3964  
3965     acquire(&idelock);  
3966  
3967     // Append b to idequeue.  
3968     b->qnext = 0;  
3969     for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  
3970     ;  
3971     *pp = b;  
3972  
3973     // Start disk if necessary.  
3974     if(idequeue == b)  
3975         idestart(b);  
3976  
3977     // Wait for request to finish.  
3978     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){  
3979         sleep(b, &idelock);  
3980     }
```

And, or I mean either it should be not valid, or it should be dirty basically ok. So, it should be either a read request write request. So, first thing you do is you acquire the idelock that is mutual exclusion the next thing you do is you append the buffer to the idequeue. So, idequeue is some shared structure as the global variable and you just

append the block to the idequeue you append it to the end of the idequeue because it is a FIFO. So, you append to the end of the idequeue.

So, you just iterate over the idequeue till you reach the end and then you append to the idequeue alright. So, just in this figure just go till the end and then put your new buffer here alright. Finally, we check if idequeue is equal to b. So, b is the buffer that was the argument to this function. So, this is the buffer that I want to read or write if idequeue is equal to b is checking what.

Student: (Refer time: 30:47).

If I am the first element in this queue that is what it means right; if I am the first element in this queue, then start the start the disk alright. So, basically it means that the queue was actually empty right now and this is the first request has been made to the disk. So, disk is actually not spinning right now it is not working right now. So, I need to start the disk. So, there is this function called idestart that is going to use in and out instruction to basically tell the disk to start there is some standard which is basically telling it to start.

(Refer Slide Time: 31:14)

```
963     panic("iderw: ide disk 1 not present");
964
965     acquire(&idelock);
966
967     // Append b to idequeue.
968     b->qnext = 0;
969     for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
970     ;
971     *pp = b;
972
973     // Start disk if necessary.
974     if(idequeue == b)
975         idestart(b);
976
977     // Wait for request to finish.
978     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
979         sleep(b, &idelock);
980     }
981
982     release(&idelock);
983
984
```

So, this disk has been started. If I am not the first element in the queue then I do not need to start it is already started right and if it is already started it will you know it will as we going to see it will basically just it is just serving one request and as soon as it is done serving that request it will pick the next request in the queue and so on right.

So, I do not need to do anything if it is already started it knows what to do next as the long as I have appended something to the queue I am fine alright and then I keep waiting on this condition. The condition is that whatever I wanted to do read or write if it is happened if it till it has not happened keep sleeping right. The mutex is idelock and the channel is the buffer on which you wanted to do this operation alright.

Let us look at you know may be it will become clearer if you look at what happens if the disk finishes.

(Refer Slide Time: 32:21)

```
3901 void
3902 ideintr(void)
3903 {
3904     struct buf *b;
3905
3906     // First queued buffer is the active request.
3907     acquire(&idelock);
3908     if((b = idequeue) == 0){
3909         release(&idelock);
3910         // cprintf("spurious IDE interrupt\n");
3911         return;
3912     }
3913     idequeue = b->qnext;
3914
3915     // Read data if needed.
3916     if(!(b->Flags & B_DIRTY) && idewait(1) >= 0)
3917         insl(0x1f0, b->data, 512/4);
3918
3919     // Wake process waiting for this buf.
3920     b->Flags |= B_VALID;
3921     b->Flags &= ~B_DIRTY;
3922     wakequn(b);
```

So, when the disk finishes the IDE interrupt function gets called the interrupt handler and what is going to do it is going to again acquire the idelock right once again. So, the interrupt handler; so, the disk is finished it has called the IDE interrupt handler it will acquire the idelock and it will check if the idequeue is now if the idequeue is null that basically mean I do not know why I got this interrupt it is possible that device creates a spurious interrupt. So, I just say you know it is a spurious interrupt do not worry about it and usually if I got an interrupt it basically means that they must have been something in my idequeue right that is why I was working and that is why I generated the interrupt.

But, if the device for some reason generated a bad interrupt, I should tolerate it. So, the programmer is tolerating it. Otherwise basically say you basically say that you know the whatever the top of the queue has been has been addressed has been serviced. So, you

basically move the top of the queue to it is next, you read the data using the in instruction from the disk into the buffers data b dot data and so b b is the top of the queue right.

So, you are serving the top of the queue. You check you read the data into the b these buffer you set it is flags to say that now it is valid or you know it is not dirty anymore it depending on whether the read or write request and then you call wake up on b right. Why do you call wake up on b because the process who was who made this request must be sleeping on b.?

So, now you call wakeup on b. So, that is how you basically coordinate between a process who made the request and the disk who completed the request for that particular process alright and finally, if the queue is still not empty which means there are more request to do then you restart the device for the next buffer alright. So, you know details aside I mean let us forget about how exactly the disk is being you know what is the interface you know what does insl mean and what does how does IDE start work these are all sort of very you know too much detail that we do not really need, but what is important is how is synchronization happening right.

So, there is a process who append something to the queue goes to sleep on that on the buffer that he wanted to actually read or write the disk when it goes to sleep it also releases the idelock. The disk when it finishes generates an interrupt the interrupt handler also needs to acquire the idelock because it needs to operate on the it needs to manipulate the idequeue for example, it will move the top of the idequeue to it is next point right.

So, it will operate on the idequeue. So, it is needed to take the lock and then it calls wake up to on any process that was waiting on that idequeue. So, basically wakeup make makes it runnable and so that process can now return from wherever it was good.

(Refer Slide Time: 35:39)

```
3962 // Append b to ideoqueue.
3963     panic("iderw: ide disk 1 not present");
3964
3965     acquire(&idelock);
3966
3967     // Append b to ideoqueue.
3968     b->qnext = 0;
3969     for(pp=&ideoqueue; *pp; pp=&(*pp)->qnext)
3970     ;
3971     *pp = b;
3972
3973     // Start disk if necessary.
3974     if(ideoqueue == b)
3975         idestart(b);
3976
3977     // Wait for request to finish.
3978     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
3979         sleep(b, &idelock);
3980     }
3981     release(&idelock);
3982 }
```

So, what if a disk generates an interrupt while. So, before that what if a disk generates an interrupt while I was somewhere here. So, I have started the disk and before I go to sleep the disk has finish. So, let say the disk is really fast it just finishes immediately alright.

Student: (Refer time: 35:48) the lock. So, interrupt handler cannot be evoked.

Right, so basically because I have acquired the idelock interrupts are disabled at this point this is the great example to understand why we need to disable the interrupts when we are holding a lock right. If we did not disable interrupts the interrupt handler could have run here, and bad things could have happened because I am holding the idelock this entire region the interrupts are disabled.

Interrupts get re-enabled only when you sleep, and you release the lock consequently alright. So, this entire region that interrupts a disabled alright; so, there is no; so, if an interrupt occurs what happens is that the interrupt gets buffered by the hardware and as soon as you re-enable the interrupts the hardware basically gives that interrupt to you.

Student: But that buffer is very small.

That buffer is very small you know 1 or 2 interrupts it is right. You know you probably worrying about the situation what if the interrupt gets lost and no future interrupt comes. I think we have discuss this before. So, let say you know in usually the protocol is that if a device makes an interrupt it expects for an acknowledgement from the CPU. So, if it

has not receive the acknowledgement, we will retry the interrupt. So, there is you know that kind of a protocol going on alright. So, if an interrupt occurs here no problem.

Student: Ok.

Interrupt will be served only as soon as the idelock gets released and so the interrupts get re-enabled.

(Refer Slide Time: 37:18)

```
3900 // Interrupt handler.
3901 void
3902 ideintr(void)
3903 {
3904     struct buf *b;
3905
3906     // First queued buffer is the active request.
3907     acquire(&idelock);
3908     if((b = ideoqueue) == 0){
3909         release(&idelock);
3910         // cprintf("spurious IDE interrupt\n");
3911         return;
3912     }
3913     ideoqueue = b->qnext;
3914
3915     // Read data if needed.
3916     if(!(b->Flags & B_DIRTY) && idewait(1) >= 0)
3917         insl(0x1f0, b->data, 512/4);
3918
3919     // Wake process waiting for this buf.
```

The IDE interrupt when it when it gets to run, we will try to acquire the idelock. You can be sure that if it was only one CPU then it will get this IDE idelock right. The only reason that that the interrupt handler may not get the idelock is because another CPU has is holding it, but the same CPU could not be holding this idelock because after all the interrupt got to run right.

So, the interrupt got to run that CPU could not have been holding the idelock if it were holding then interrupts would have been disabled. So, that the whole reason the whole the whole fact that the interrupt handler got to run means that the interrupt the CPU was not holding any lock. So, you know you will that basically means that you will eventually get idelock. So, there is no deadlock problem here right.

(Refer Slide Time: 38:07)

```
3911     return;
3912 }
3913 i dequeue = b->qnext;
3914
3915 // Read data if needed.
3916 if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
3917     insl(0x1f0, b->data, 512/4);
3918
3919 // Wake process waiting for this buf.
3920 b->flags |= B_VALID;
3921 b->flags &= ~B_DIRTY;
3922 wakeup(b);
3923
3924 // Start disk on next buf in queue.
3925 if(i dequeue != 0)
3926     idestart(i dequeue);
3927
3928 release(&idelock);
3929
3930
3931
3932
```

And, then you will perform this operation and then you will wake up the particular process. When you call wake up that process may not necessarily run immediately; let say it was a uniprocessor system this interrupt handler is running. So, you know how can that process run you have also disabled interrupts completely here, but when you release the idelock and then you return from the interrupt, the interrupts get enabled again and you know because you have woken it up it has become runnable whenever the schedule gets run next it is going to get picked up and it will proceed alright.

So, let say the process wakes up from sleep, it checks the condition again this time it finds it to be true false and no it can go on it is way alright. Notice that here inside the loop when I am saying while some conditions sleep on this. I have not really checked for p dot killed right. I said that usually when you come out of sleep you should now because you are doing this thing that any sleeping process will be made runnable is it incorrect to not check for sleep dot p dot killed right.

Student: (Refer Time: 39:10).

Well I mean it does not matter if it is going to check the condition even if the process was killed you still want that the buffer has been read lock has been released and now you can go on and check later alright. So, it is in this case. So, it is not necessary that every sleep loop needs to be needs to have the killed condition check killed check you

know some need some do not example and one has to reason carefully about what means it and what does not.

Student: Sir, why do we needed in the space.

(Refer Slide Time: 39:49)

```
968     b->qnext = 0;
969     for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
970     ;
971     *pp = b;
972
973     // Start disk if necessary.
974     if(idequeue == b)
975         idestart(b);
976
977     // Wait for request to finish.
978     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
979         sleep(b, &idelock);
980     }
981
982     release(&idelock);
983 }
984
985
986
987 NPTEL
988
```

So, one way to think about it is that eventually it is going to come out right. So, it is not it is not a very long time that it is going to stay there. Now, I am not waiting for some child to exit for example, I am just waiting for the disk to finish alright.

So, it is better to you know let it finish and then release the idelock and then basically let it is caller call. So, make let the things be in a more consistent state you know. I do not know whether the caller is leaving things in an inconsistent state or not. So, you know let it reach some boundary this may not be the right boundary this may be the innermost sort of thing and if I just sort of started returning minus 1 from here that is not that may not be the right thing to do.

So, just an example sometimes you may want to check it sometimes you may not want to check it, but the thing is that within a limited time frame you should basically be exiting that particular process alright. Here is another this example is also another an interesting illustration of why recursive locks are bad idea right.

So, recursive locks are bad idea here why because notice that mutual exclusion is required between the thread and the interrupt handler right and the mutual exclusion is

being done using the idelock instead of you know instead of making idelock like the wait is if I made them made the idelock recursive and the interrupt handler was actually able to get the idelock also then bad things can happen because you know you can get take a lock you could be middle of something and now in the interrupt handler gets called now it takes the idelock and so it and it assume some invariants which are not guarantee to be true right.

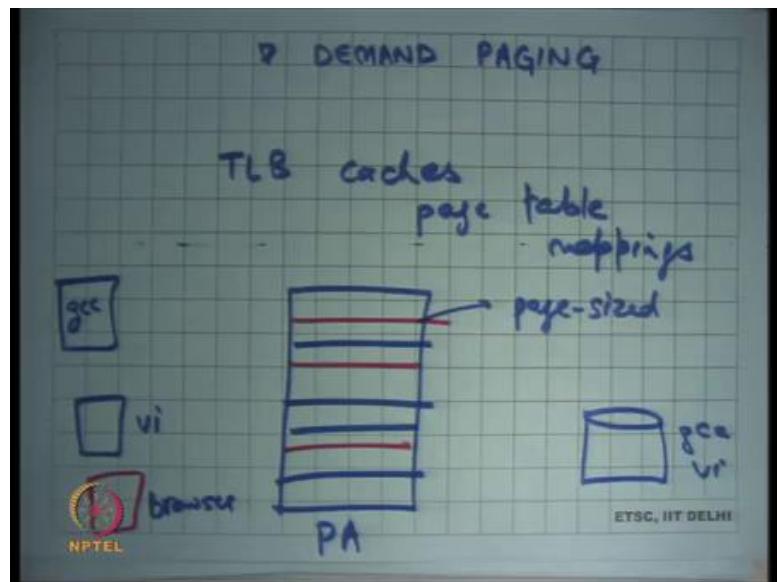
So, recursive locks specially for in presence of interrupts are definitely a bad idea in general also you know it encourage bugs, but if it if the you know you are providing mutual exclusion with respect to interrupts then I then recursive locks are a bad idea alright. Also, because I have you know I am holding a lock I am trying to acquire idelock I need to basically ensure that there is some kind of ordering.

So, any caller if the it is possible that the caller also holding certain locks. So, this idelock needs to be ordered with respect to whatever locks the caller is holding. So, you know one invariant could be that the idelock will always be the innermost lock you will never hold idelock and then try to acquire another lock. So, that way you can prevent deadlocks right.

So, you have to worry about these things also alright because; so, another example of why locking and modularity do not yeah because, yeah you have to worry about ok. So, I am taking idelock here I should not be taking idelock somewhere else where it is possible that you take the idelock and try to take idelock somewhere you know another lock after that good.

So, with that I think you know we have done lot of synchronization and synchronization was you know by far one of one of the most sort of important practical topics in programming in general and in operating systems.

(Refer Slide Time: 43:04)



Now, I am going to talk about demand paging alright. So, we know that we know we have we have looked at virtual memory and we saw virtual memory using segmentation where there was a base and limit we also saw virtual memory using paging and we said you know paging is more flexible. Although paging has more overhead because you have lots of page size you need to maintain a page table and so, but we said then there is a cache called TLB right.

So, there is a TLB that caches page table mappings right and assuming that this cache has a very high hit rate and this cache is very fast by fast I mean it is you know sub nanosecond. So, and the time it takes to actually dereference the TLB is roughly equivalent to the time it takes to dereference that is just right. If it is that fast, then you know paging makes a lot of practical sense

And so, you know hardware developers. So, TLB is actually work in practice because usually programs have a lot of locality lot of spatial and temporal locality which means that the same locations are likely to be accessed over and over again also if you access the location then very likely you are going to access locations close to that. So, because programs exhibit a lot of spatial and temporal locality the hit rates of TLB caches are usually very high and you know on the order of 99 point let say 9 percent or something and so you basically you know do not pay the cost of paging that right.

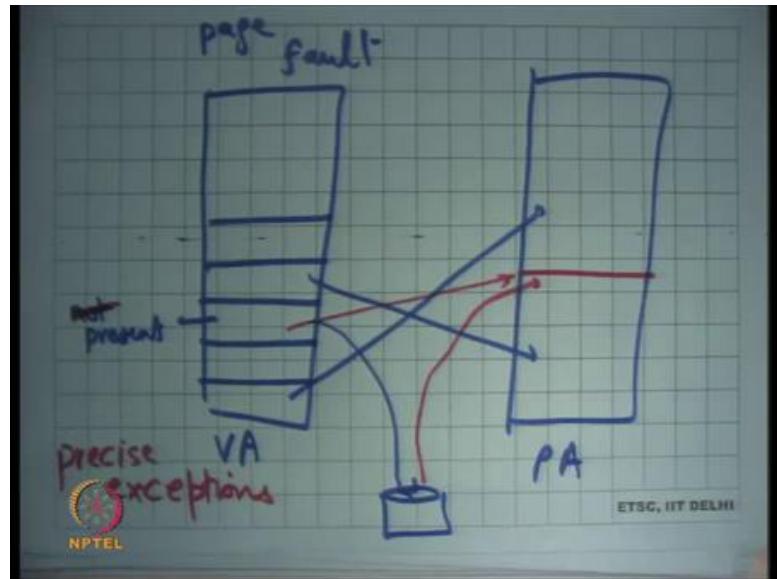
So, the cost of paging was basically this dereference of page table, which is costly so, but that gets eliminated because of TLB cache and so usually. So, so here is how your memory will look like. So, let say this is the physical address space and these are multiple virtual address spaces let me draw this with the different color let say this is gcc and this is let say vi and and so on this is browser then some pages the pages are strewn across like this in the physical address space ok.

And these lines are page sized ok. So, we understand all this. Now in the simple word that we are discuss. So, far we said that whenever you load a program the entire memory of that program the executable and whatever other data it needs is basically loaded from the disk into the physical memory and load time right. So, basically initially the program lives on disk let say gcc and vi including it is code and data and as soon as you load it we say that the entire you know there is some format called a dot out executable.

And, so the a dot out format is fast and the loader basically creates an address space, allocates pages in the physical address space creates an address space and loads the entire contents of the executable into physical memory alright. It is quite possible that you know the process was just started to just stop immediately or it is not going to access all that memory that it actually has in the executable.

So, it is going to access only your fraction of the memory that is going to do it. So, it does not make sense to actually pull all the things at once and so what you can do is you can pull things from disk on demand and that is what is called demand paging right. So, the idea is that you basically at load time you just create an address space alright.

(Refer Slide Time: 47:02)



So, let say this is an address space. This is the virtual address space and let say this is the physical address space and let say this is the disk then you have created the address space. Let say these you know these slots have pages. So, some of these pages are currently mapped inside your physical address space and others are not currently loaded and are actually pointing to the disk at some disk block right.

So, it is not actually a loaded. So, the page is not loaded immediately, but in the page tables you have stored this information that this page corresponds to this particular disk block alright. So, and you say that it is not present alright. So, this particular page is not present. So, the present bit is off in the page table and you have stored this information that this particular page actually lives on the disk at this particular offset.

So, when you run this process you are going to you know if you do not touch this page and you exit you know you have saved a lot of work. If you touch this page, then you basically take an exception right. An exception happens because you try to access a page that is not present right and so this exception is called a page fault. If you try to access a virtual address that is not currently mapped in the page table or it is not currently present in the page table, you take an exception that is called a page fault.

The page fault will cause the operating system to run the operating system page fault handler to run on the kernel stack of that particular process and in the previous discussion you were saying that this the page fault handler may want to kill the process

or it may want to send a signal to the process depending on whether the operating system implement signals or not. But, in this case you may want to do one more thing which is to check if this page is actually mapped to a disk location and if so it should not do any of these, it should not kill the process. In fact, you know this is operating system playing tricks under the carpet the process was actually you know doing everything in good intentions if the operating system that is playing tricks under the carpet.

So, what it should do is it should allocate a page here. Let say it allocates a page here loads from the disk block to here marks this present and create some mapping like this alright. So, that is called demand paging and demand paging is a very very sort of useful optimization because you know for the common case you do not need all parts of the executable are not going to get accessed only some parts of it is going to get accessed.

And, so you save a lot of disagrees alright and also you reduced the pressure on your memory you do not you do not need to allocate that many pages on your memory. So, there is more free memory available on your system right. In general, you know your system may be running only on a small amount of memory let say your system is running on 512 megabytes of memory, yet you can your operating system will allow you to load larger processes.

So, for example, your operating system will allow you to load you know gigabyte size process on a machine of size 512 MB. It is done because of the demand paging running underneath the covers right yeah.

Student: Sir a page fault occurs, and we load the missing page (Refer time: 50:27) into the pageble now we want to re-do the command which caused the page fault how is it.

Yeah, good question we are going to discuss it very soon. So, basically there is some instruction that try to access that particular address and so a page fault occurred. The operating system is going to get to run and it is going to load the page and now you will need to restart that instruction alright and so that is how it is run basically you just restart that instruction.

What you need to make sure is that if you run the instruction twice or you know the first if an instruction causes an exception it does not cause any partial execution of its logic

right. So, either the instruction completes successfully, or it does not do anything at all and causes an exception alright.

So, this has an, this is the property of the hardware or the architecture and this property is called precise exceptions alright. So, if an architecture supports precise exceptions, if there was an exception at an instruction it is safe for the operating system to restart that instruction and basically assume that nothing happened in the previous execution of that instruction alright.

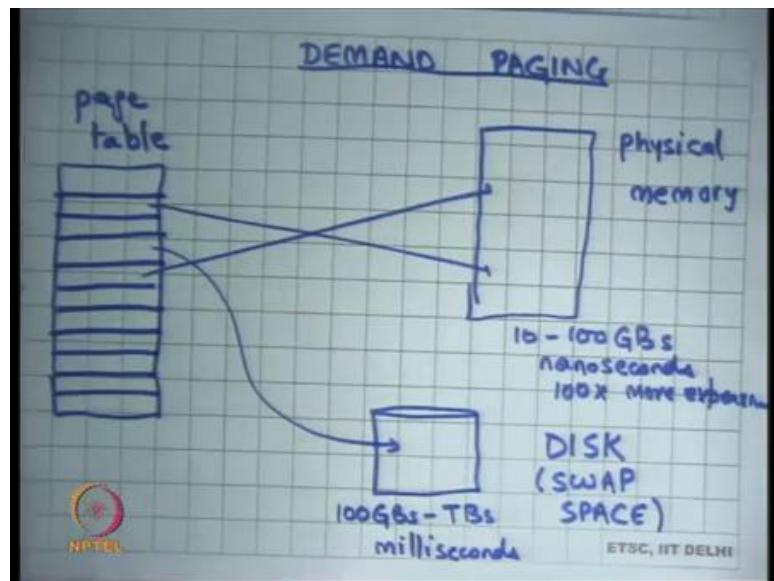
So, we will continue this discussion next lecture.

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

Lecture - 29
Demand Paging: Introduction to Page Replacement

Welcome to Operating Systems lecture 29 right.

(Refer Slide Time: 00:29)



So, we were discussing Demand Paging last time and we said that demand paging is a way for the OS to fool the process into believing that it has more memory than it actually is occupying in physical sense right. So, a process may see a very large virtual address space and recall that the virtual address space is implemented by a page table assuming you are using paging.

And so, it may see a very large virtual address space, but only some part of it may actually be present in the physical memory, other parts of this virtual address space may be present in the disk alright. And so, if this is the page table and these slots are the pages in some of these pages will have mappings to the physical memory and some of them will not have mappings to the physical memory and logically speaking they will have mappings to the disk alright.

The you know this area of a you also where you store the virtual memory pages in the disk is also call the swap space alright or the swap partition if you may have heard on different operating systems alright. But it is not necessary that these pages point to the swap space, swap space is basically extra space reserved in the disk.

Student: (Refer Time: 01:46).

To allow pages to get saved from physical memory. So, and a so on, but these pages may actually even be mapping into the file system right. For example, if you loaded a dot out file then some pages have not been loaded at load time and so, these pointers will actually will not be point into the swap space, they will be pointing to the file system space in the disk. In either case basically these pages from the hardware standpoint do not exist right.

And so, if a user ever executes an instruction that tries to an access a virtual address somewhere here, it will result in an exception and this exception is called a page fault right. So, page fault will occur and as we know if a an exception occurs control transfers to the operating system. And, the operating systems page fault handler will get to run; without demand paging the page fault handler would have probably just scaled the process.

But, if the operating system is implementing demand paging then the page fault handler should additionally check if the faulting address or the address on which the page fault was generated, if that actually has been points to the disk. So, internally the operating system is maintaining some data structure which basically says you know which pages are mapping to the disk and which pages are not present at all right. So, if this page is actually mapping to the disk then the operating system will load this page from the disk to the physical memory.

And, then create a mapping from the page table to the physical memory and restart the instruction right. Now, restarting the instruction needs to be safe as we had discussed last time and we you know typically; so, one way to do this is basically to rely on the hardware to provide certain guarantee.

So, one guarantee that is provided by the hardware is that if an instruction generates an exception and page fault is one example of an exception then before the exception

handler is called the machine state will appear as though the instruction did not execute at all right. So, it is not like the instruction is partially executed. So, the page even though the instruction generated an exception in the middle of with execution the its a responsibility of the hardware to basically rollback all execution state of that instruction before it invokes the exception handler.

So, all the instructions previous to that instruction would have executed and that instruction will not have executed at all and of course, no instruction after that would have executed. So, now the page so that allows the operating system to just restart that instruction. And, how does the operating system restart the instruction? Just by putting the return address or the in the PC, in the tab frame to the instruction that faulted right.

And so, when you when the operating system does irate just goes back to the same instruction and that instruction gets run again. Because, the page table mapping has been created by the operating system this time this particular instruction will not fault right and so, it will continue.

And so, this is demand paging the idea. So, using this demand paging mechanism and operating system basically in some senses using the physical memory as a cache for the disk. So, you can imagine that this is you know one way to think about it is that the entire physical memory pages are actually on the disk, but the physical memory.

So, the entire virtual memory pages are actually on the disk. So, the physical memory is acting as a cache to that space right. So, it just that some pages have been loaded into physical memory and so, you know you basically.

So, most of the so, which ever pages have been loaded to the physical memory execute at full speed, any pages that have not been loaded into the physical memory or the cache have to take a page fault and you basically access the disk in that case. Let us understand some hardware characteristics of physical memory and disk to understand you know what are the tradeoffs involved in this caching setup.

The physical memory is you know one common technology is D RAM meta you today you will get 10 to 100 GBs of 100s of GBs of a DRAM. But, its more expensive its usually 100 times more expensive than disk ok, 100x more expensive roughly speaking.

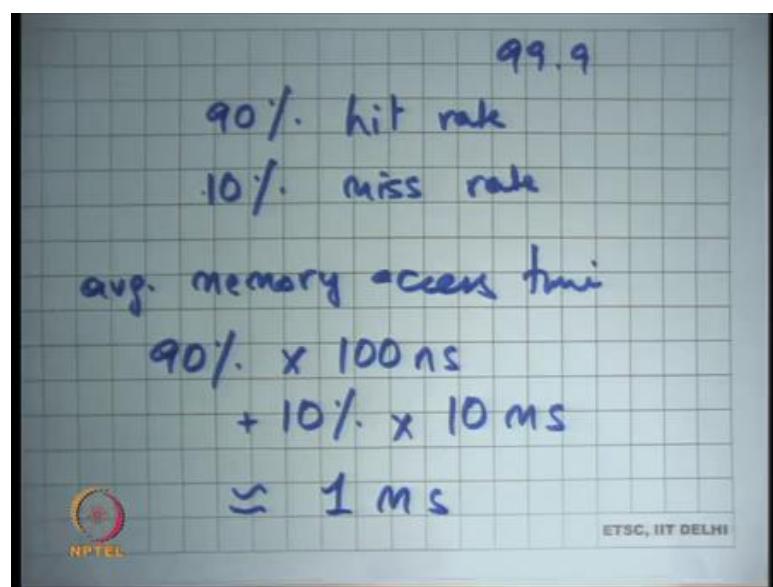
And, disk on the other hand you can get you know 100s of GBs to terabytes and so, but it is much cheaper.

So, you can you know for the same so, byte cost per byte is 100 times more in physical memory then it is in disk. So, you can afford to have much larger disks then you can have physical memory. The more important trade off point is that a disk access takes milliseconds to access while a memory access takes nanoseconds to access.

So, there is a you know 10 to the power 5 or 10 to the power 6 performance difference between these things alright ok. So, you can imagine that if there is such a huge; so, basically a cache hit is only going to take 10 to 100 nanoseconds let us say, but a cache miss is going to take millisecond.

So, you know miss cost is much higher than hit cost and so for this scheme to be successful you would want that the hit rate is very high alright. If the hit rate is small then you would basically be always you know you will be executing the speed of the disk.

(Refer Slide Time: 07:17)



So, let us take an example let us say you have a hit rate of 90 percent alright. So, 90 percent hit rate ok; so, let us say I had a 90 percent hit rate and 10 percent miss rate right. So, what is the what is my average memory access time? Average memory access time

would be 90 percent into let say 100 nanoseconds plus 10 percent into let say 10 milliseconds right which is roughly equal to actually this cost is almost 0.

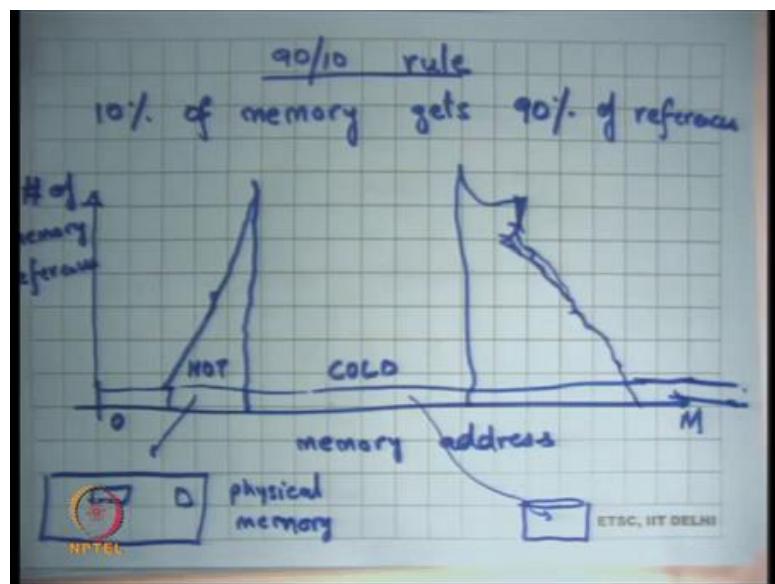
This is actually 10 percent into 10 millisecond that is roughly equal to 1 millisecond right. So, on average every memory access is going to take 1 millisecond of time right that is a very bad; I mean your actual physical memory was actually only 10 to 100 nanoseconds. And, now you know if you have; if you have a 90 percent hit rate you are basically executing at the speed of the disk, completely unacceptable right.

So, these kind of numbers are absolutely not acceptable, what you would; what you would be perhaps with is something like a 99.9 percent hit rate or something right. So, you know less than 0.1 percent miss rate and so, that will basically give you the illusion that yes you have the space of the disk, but the access time of memory alright. So, really the OS is trying to.

Student: Give.

Give the illusion that you have the space of the disk, which is you know 100 GBs to TBs, but the access time of memory right and fortunately in practice it is possible to do that.

(Refer Slide Time: 08:51)



And, it is possible to do that because of this 90 10 rule I mean also call the 90 10 rule which basically says that this is a high amount of locality in accesses by a typical programs. So, programs typically 10 to access the same memory locations over and over

again. So, that is called temporal locality, also programs tend to access locations close to the ones that they have accessed. So, if they access location x then quite likely that it will access either $x + 4$ or $x - 4$ so, that is called spatial locality right.

So, programs exhibit a lot of spatial and temporal locality and so and so caching the whole the whole basis of caching is that there needs to be some locality right and because and there is a huge amount of skew. So, basically the 90 10 rule says that roughly speaking typical programs 10 percent of memory gets 90 percent of the references.

So, if this 10 percent of memory is brought into the physical memory then you know you already have a very high hit rate. So, 10 percent of memory gets 90 percent of the references. So, if I were to just plot a graph which says you know let say this is the memory address 0 to let say whatever is maximum virtual address you have.

Then you know and you were to plot what memory access addresses are accessed how many times. So, on the y axis you are basically saying how many times was this memory address accessed throughout the life of the program. Then you know you have some small thing here which is saying that these memory access were accessed sometime once or twice or thrice or 10 times or whatever.

But there is some memory access says which have a huge number of accesses right. You can what are these memory addresses likely to be? Let say the code pointer right. So, if you are executing the program in a loop the same EIP or the same program counter is going to get accessed over and over again right. What is the stack pointer?

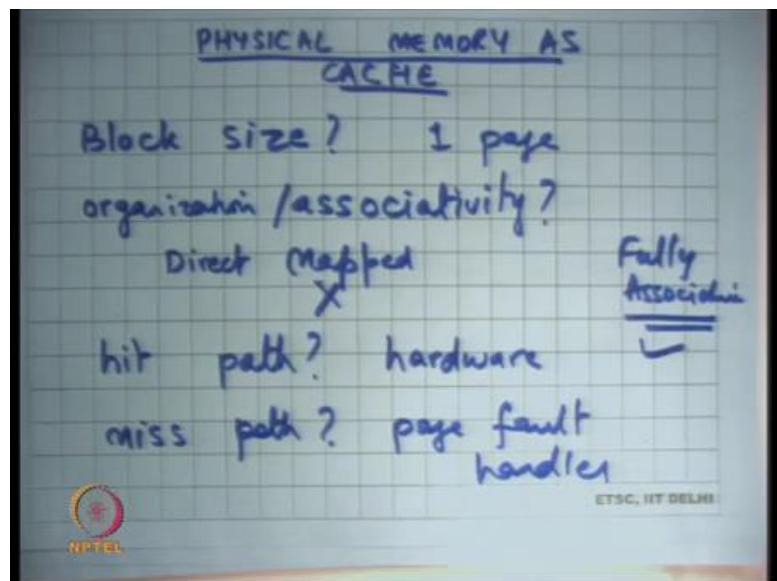
It is likely that the stack point is basically the stack pointer is moving in the same range. So, you basically accessing the same page over and over again right or even data. So, you know even the data structures of the heap, it's quite likely that you know the root of let say you are maintaining a tree in your program; let say binary search tree or something.

Then the root of the tree is likely to be very hot you are going to access that root each time you do anything on the tree or something right. So, in general there is a high amount of locality and you know the higher this bar the basically shows you know temporal locality because you know its same location is being accessed logarithms.

And, the spread of it can be you know called spatial locality because few of accessing this few are also accessing somewhere something close to it. So, it sorts of a sort of seems to have this kind of behavior. So, an operating system would be doing its job correctly with respect to demand paging, if these pages which are very hot are mapped into physical memory and these pages which are cold are mapped into addresses.

So, you can also call these addresses cold addresses and these addresses hot addresses right. And so, the hot addresses should be mapped into physical memory and the cold addresses should be mapped into disk. And, if you can do that then you basically have a system where you have the speed of physical memory and the size of the disk alright ok.

(Refer Slide Time: 12:11)



So, we are saying that the physical memory as cache right. So, one way to think about this entire demand paging scenario is to think of the physical memory as a cache, as cache to the disk which is much bigger. So, clearly you know any time we will talk about a cache we basically know that the cache has to be smaller than the entire data set. And so, whenever you talk about a cache you have to worry about you know somethings.

For example: what is the block size? It by block size; that means, what is the unit of data which is cached where is the smallest unit of data that is cached. For an L1 cache for example, it will be your cache line which is you know 60 into 64 bytes depending on your architecture. But, for this what is the block size?

Student: 1 page.

1 page right. So, 1 page alright. How does one choose the right block size? Well, it should not, if the block size is really big then you may be getting too much extraneous data. So, let say in this figure you are accessing address x and your block size was let say 4 megabytes, then you are going to you know you are going to bring in x to x plus 4 megabytes into the memory. And, it is quite possible that you know all that data is not very cold, it just that x and x some data round x was hot and everything around it was cold.

So, you are basically polluting your cache with cold data. So, if you choose a very large block size you have; you have; you have that problem that you are basically having cold data sharing. On the other hand, if the block size is really small then the firstly, you will heard spatial locality right.

So, let say x and x plus 4 are very likely to get access together, but because your block size is very small you do not get x plus 4 you just get x and so, for x plus 4 you take an another page fault right. So, that is one thing, but more importantly recall that a disk has a very long sort of access time right.

And, also the access time is basically dominated by the seek and the rotational latency right. So, it is better that if you are paid that price you get a large chunk of data from the disk at once rather than just getting 1 byte to get a larger chunk of data from the disk.

And so, you know there has to be some tradeoff between these two constraining forces and the 1 page seems to be a reasonable value and we will discuss more about this a later. The other thing is what is the organization of the cache? Right by organization I mean associativity right.

So, you know in your computer architecture course you must have studied, and your cache can be direct mapped, set associative or fully associative right. So, what you think makes more sense in this case? Should you have a direct mapped cache, or should you have a fully associative cache? . So, let us see you know what is a tradeoff? You know let say direct mapped versus on one side and fully associative on the other side right.

So, in direct mapped I basically know that here is a location for you know these set of addresses. And so, when you get a miss the one that you replace is the you know it is very clear whom you are going to replace. This is the you know there is a conflict between the addresses of the one that is present and the one that is access and the one that is present and gets replaced and the one that is being accessed gets to run right.

So, in other words you know replacement is very fast alright, but the price you pay is that you may get extra misses, conflict misses. Even though you could have adjusted both pages in your memory because, you are doing direct mapped you are getting more conflict misses right. On the other hand, fully associative has full freedom in choosing the page that needs to get erected right.

So, in this scenario where the miss cost is really high and maximizing your hit rate is the for prime importance and it cost of actually finding the page to replace is not going to be very high compared to the miss cost right. How many instructions will need to be get executed to find the page the to find the best candidate for replacement, its only some memory accesses after alright.

And, whatever those memory accesses are you can actually execute million memory accesses before you actually even reach close to the miss cost right. So, you know because the cost to actually find the replacement element; even in a fully associative cache is not very high compared to the miss cost.

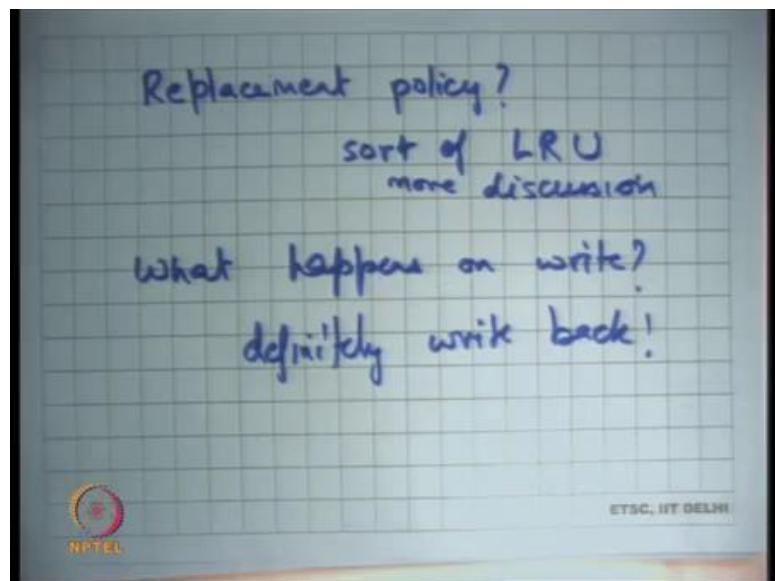
And, because hit rate is of prime importance, it is a fully associative cache makes make sense right as supposed as set associative or a direct mapped here alright. So, let say you know how does one; so, what is the hit path? Well, we want that the hit path should be released fast which means how does one how does the program, how does the system check whether the access as the hit.

Well, the a hit actually has no software involvement, it goes purely in hardware. So, an instruction executes, it has a memory address; the first thing you are do is you check the memory address, translate the memory address using the TLB right. If you hit from that you have just done the translation in 1 nano second; so, that is the; that is the fastest hit you can have right that is 1 nanosecond.

If you do not hit the TLB then you do the page table walking, page table walking will be 10 to 100 nanosecond. So, that is the second level hit in some sense. And, if you hit if you miss there then you do the disk access which is milliseconds alright. So, hit path is basically completely implemented in hardware and miss path will require software involvement because, the miss path will require the execution of the exception handler alright.

What is the miss path? Well, miss path is page fault right and this is much slower alright. And, let see what have what is the page replacement policy or a replacement policy, every cache must have a replacement policy right.

(Refer Slide Time: 18:37)



So, what is the replacement policy? Basically, you want to bring a page into the cache, and you have to figure out and your cache is already full. So, you have to figure out one page whom you are going to replace and as I said its very important to maximize hit rate. So, what should be the replacement policy? You know it should be some kind of fully associative policy requires more discussion, but sort of LRU; least recently used.

You have seen least recently used before in computer architecture or something alright. So, sort of LRU and we are going to discuss this more alright. Finally, what happens on a write? So, recall that a cache has needs to have a write policy. So, if you write something should it be a right through, which means it goes all the way you know on every write you go all the way to the bottom or should be a write back?

Student: Write, write, write.

Write definitely, write back right because if its write through then every write is going to access, we access a disk space; so, definitely write back ok. Now, alright so, let us see how does the operating system implement this. So, block size 1 page this is handled by the page table, associativity is fully associative, this is you know this is the page this is handled in the software by the page table handler.

So, this is wrong, and this is right, a hit path is hardware. Now, but in the hit path you also need to so, the hardware should also tell whether it was a hit or a miss right. So, the hardware is telling whether it is a hit or a miss using what?

Student: Page fault.

Using the a page fault. And how does it know whether it needs to generate the page fault? By looking at the page table entry and the.

Student: (Refer Time: 20:43).

Present bit in the page table entry. So, that is the way you need a fast way of figuring out whether it is a hit or a miss and the fast way is basically for the software to set the present bit and the hardware to read the present bit. So, basically the present bit in the page table entry is helping you to figure out whether it is a hit or a miss and you need to do it fast.

So, it is happening in hardware alright and miss path is the page fault handler. Replacement policy is implementing in this, but what happens on a write? Definitely write back, how does you know how does the hardware indicate that this you know this page is actually needs to be written back alright.

So, how does the how can the hardware indicate? So, basically if it is a write back policy the program should execute at full speed and it should just keep writing to these pages. And, at cache replacement time you will basically figure out whether this page needs to be written back to the disk or not.

One conservative policy is that whenever you replace a page you always write it back right, that is unnecessary because it is quite possible that the page was only read from

and never written too. And so, the contents have not changed so, you do not need to write it back and.

Student: (Refer Time: 21:50).

So, you can save disk accesses and recall that disk saving disk access is a huge optimization. So, what is some other way of letting the hardware to let us know whether this page has been written to or not or modified or not?

Student: Maintain a dirty bit.

Maintain.

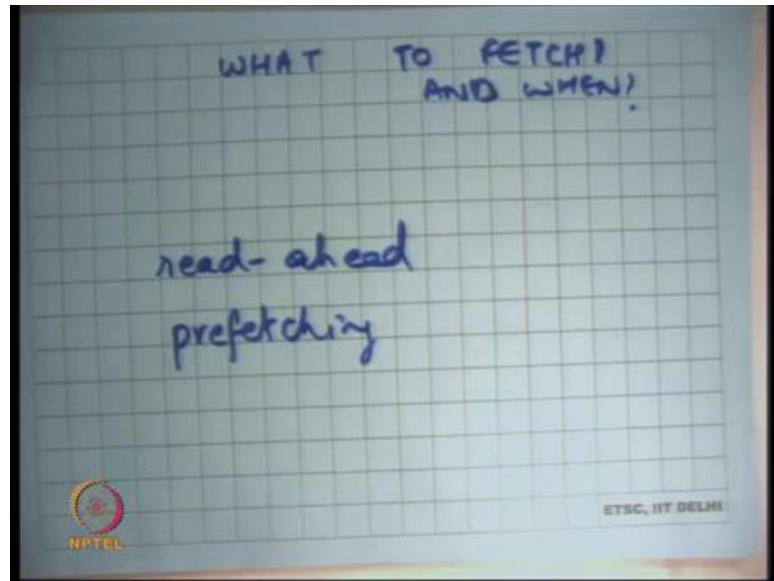
Student: (Refer Time: 22:06).

A dirty bit in the page table. So, one option is so, what is done on x86 is that the page table entry has another bit called the dirty bit. The operating system when it brings the page into the cache clears the dirty bit and if the page is written to the hardware sets the dirty bit. Notice that because the setting of the dirty bit is on the hit path it should be fast and its done by hardware ok.

At cache replacement time the replacement, the page fault handler will basically look at the dirty bit to figure out whether it need to write it back to disk or not write it back to disk. The dirty bit will be set by the hardware, it will be cleared initially by the software because when it brings the page in it will be cleared.

Because, at that time the page is clean and by clean, I mean that the contents of the page are identical to what they exist on the disk right. So, as soon as you bring the page into memory its clean, if you if the program ever writes to that page the dirty bit and the page table entry get set alright right.

(Refer Slide Time: 23:19)



So, now let us also look at what to fetch and when alright. So, for example, what should I fetch at load time? Right. So, this is the design question that an operating system designer has to figure out that you know at load time, if you basically say I want to run this process what pages should be fetched into physical memory a priori.

One option could have been that I just ask the user. So, you will provide an interface between the user and the operating system. For example, have an extra argument into your exist system call saying this is the list of pages that you should load apriori right. This is possible, has been tried, but the problem with this is number 1 you do not want to trust the user and the user may say you load all the pages.

And, you know you do not want to really satisfy his requires because that can lead to other process is having poor performance. Secondly, the user can user himself actually does not know and does not want to care about this right. When you write a program you do not care about what pages are hot and what are not, you basically want to basically say that you know here is my program you done it and you figure out what is hot and what is not and I will just manage things efficiently for me right.

So, asking the user is both sort of not very trustworthy and its it complicates the interfaces and makes puts more responsibility on the shoulders of the programmer than its actually really needed. The other thing is you could load some initial pages that load

time, you can say that you know whatever is the first program counter that needs to with executed I will load one or two or three pages around that.

So, some kind of heuristic policy around that, then you load some pages which are for your stack and maybe some data pages based on some heuristic which in operating system is free to choose and different operating systems will have different variants of this. And, then as the program execute its going to access the memory and then implement demand paging.

Each time there is a page fault once again you have a choice, do you only fetch the page that was faulting that was faulted upon or do you fetch some pages around that also. So, you can say you will have something called read ahead which is basically for spatial locality. So, whenever you fetch a page you also fetch some pages around that page.

So, that is called read ahead alright and that is basically that is going to help if you are likely to have a lot of spatial locality in your program. And, most operating systems will do some sort of read ahead somewhere right and when to do read ahead and then not to read do read ahead is also a bit of an art. And, you know many heuristics are present in real operating system to figure out this.

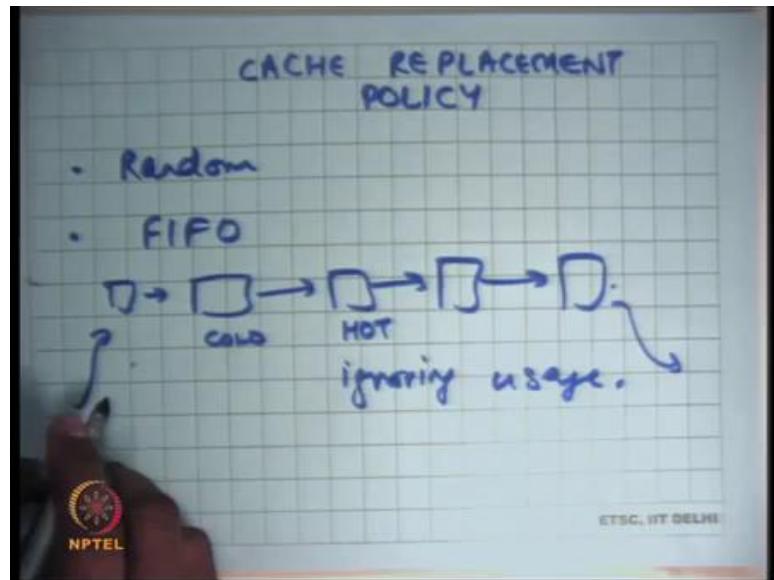
What is the tradeoff of doing too much read ahead? You may be polluting the cache with unnecessary data. And, what is the; what is the what is the disadvantage of doing too little read ahead? You may be encourage too many page faults unnecessarily right. So, that is basically also you know you can do what is called prefetching.

So, this is for temporal locality, if you basically say that every time you access page x as see as notice this pattern that every time you access page x you also access page y. So, if the operating system under the covers has been noticing this pattern in your program then it will basically say ok.

You know each time you take a page fault on page x let me also prefetch page y and this can happen for example, you know whenever you access this particular code region, this particular code always accesses this particular data structure right or there can be many such example; I am just giving you one example right. So, notice that you know the operating system is inferring all these performance decisions under the hood without the programmer having to know anything about it.

And, that makes the programmer you know really carefree about these things and that is a nice thing and it has worked for many years now and it has some limitations. However, and for example, in modern hardware like multi core you know this kind of inferences are harder and harder to make. And, all kinds of ideas have been proposed and on how to do this better with or without programmer involved it alright.

(Refer Slide Time: 27:13)



The other thing is cache replacement policy right. So, what page to evict? So, you have figured out that this is the page I want to bring in or this is the set of pages I want to bring in. So, if you have decided that these are the set of pages you want to bring in, what are the pages you are going to throw out or reject right?

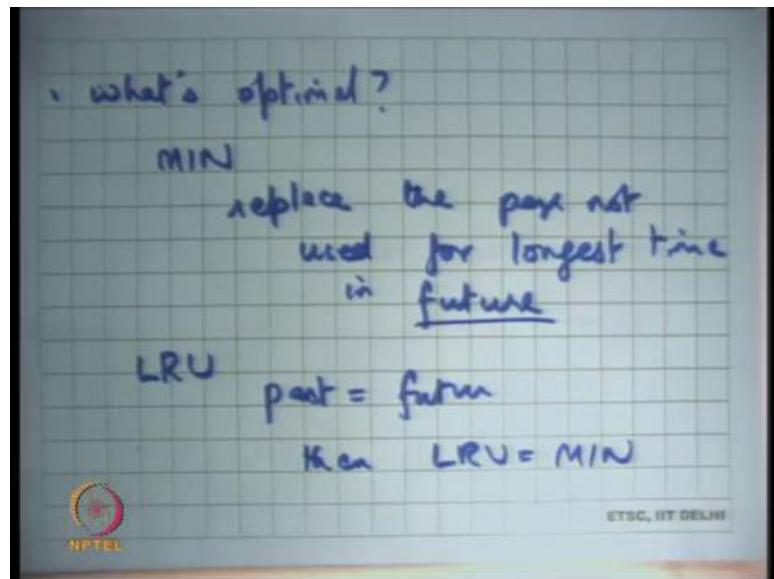
So, the first policy is random, just pick up any page or any set of pages in the thing and just in your cache and just eject them. The advantage of random is that it is very simple, you do not need to maintain any data structure.

The disadvantage is that you may actually be ejecting hot pages and that is not a good thing right. The other approach is FIFO whichever page has been brought in last you know earliest should be ejected. So, basically the way to implement it is basically have a cube in your memory, it's basically says this is you know every time you bring in a page you append to the cube. And, every if you want to eject you eject from the tail of the cube right.

Once again the nice thing about FIFO is that, it is very easy to implement; each time you bring a page in you add just to the data structure, adding to the data structure is cheap with respect to the full cost of the page fault handler, you are also making a disk access. So, you know just adding to the list is very easy, also you know removing from the list is very easy, it's very cheap. You know you just anyway ejecting the page you probably going to have a disk right or anyways taken a page fault; so, you know ejecting is easy.

So, but the disadvantage is that this organization completely ignores the usage. So, let say this page was hot and this one was cold right and you bringing in some page. So, because its FIFO the hot page gets removed and the cold page remains; so, you are ignoring usage alright. So, let us you know let us turn it around and let say what would have been the optimal.

(Refer Slide Time: 29:23)



So, let say what is the optimal page replacement policy? Assuming I have an oracle who, can tell me you know who can give me every all the information I need; what would have been the optimal thing to do? So, the optimal thing would have been that if I am bringing a page in I want and if I have an oracle who can tell me what the future is going to look like; the I can ask the oracle what is the page that is likely to be used farthest in the future you know.

So, of all the pages that I have what will page that is going to likely to be used farthest in the future. So, that is the page I want to replace because all the pages, all the other pages

will be used before that page. And so, you know you want hits on those pages and the one that is farthest that is the one you want to replace right.

So, let say let us call this policy the minimum policy which is basically replace the page not used for longest time in future right. Of course, I am assuming that there is some oracle that is telling me what the futures going to look like, and you know this is impractical. I do not know what the future is going to look like which pages are going to get access, depends on what paths the program takes right. And, I cannot predict as an operating system what paths the program is going to take.

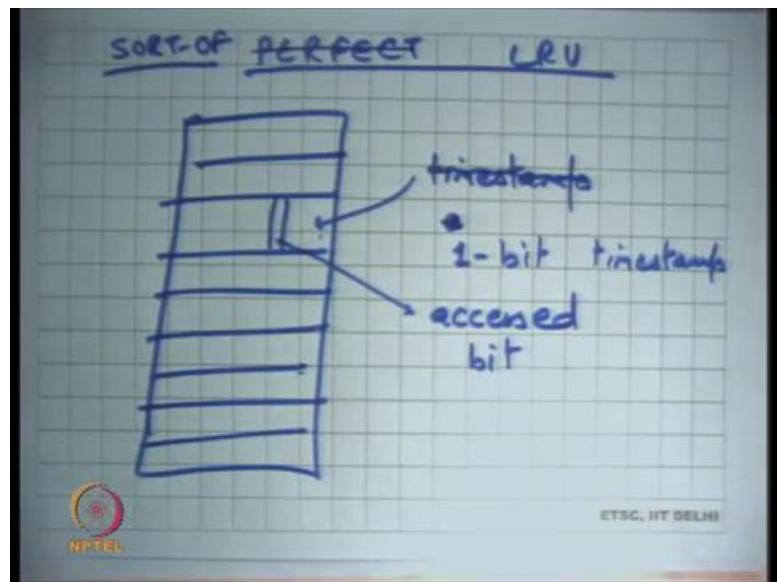
I can guess them, I can ask the user for them, but these are all sort of just heuristics or and asking the user is actually very error prone; so, you do not want to do that. So, LRU is an approximation to so, assuming; so, one common principle design principle is that if you do not know the future assume that future will look something similar to your past.

So, what is just happened in the past, the future is going to look something similar to that alright. So, if I were to if I were to sort of make this assumption that past is equal to future then instead of saying replace the page not used for longest time in future, say replace the page not used for longest time in past.

So, whatever is the page that is not been used for the longest time in the past that is the page you will replace right. So, just replace past with future, if past is equal to future then LRU is equal to MIN, if past is roughly equal to future LRU is roughly equal to MIN alright; so, that is a idea ok.

So, we looked at how random is implemented, we looked at how FIFO is implemented; clearly you cannot implement MIN, it is a completely impractical algorithm. But you can implement LRU and let us take let us look at what it takes to implement the LRU alright.

(Refer Slide Time: 32:01)



So, one way of implementing perfect LRU is that let say you have a page table or a virtual memory space whatever you want to call it. And, these are the pages, each time you access a page you also store a timestamp with when it was accessed.

Each time you access a page you know record the timestamp at when it was accessed and then when you want to replace you go through all your pages and look for the page with the smallest timestamp alright. Now, who should put the timestamp on the page?

Student: Hardware.

Hardware right because it is a it has to be on the hit path. So, if I cannot I do not on the hit path there should not be any software involvement, I want the hit path to be as fast as possible. So, the timestamp has to be put by hardware and so, that does not sound very practical right. I mean what kind of timestamp should be put by the hardware and how big the timestamp should be.

And, each time if I have to put a timestamp then there is extra overhead. So, recall that you know each time I put a time stamp I have to make a memory write in some sense to put the timestamp and that is not; that is not great. Also, at the cache replacement time I have to go do a global sort on the timestamps and then figure out which is the least right.

The global sort may not be that much of a problem given that anyways there is a disk access involved, but you know on the hit path having to put this timestamp is not

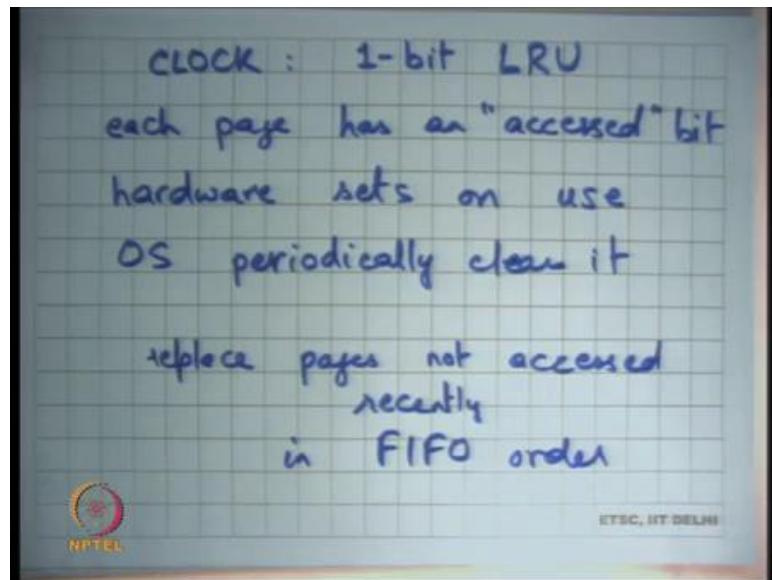
practical. So, what is done is actually not perfect LRU, but you know let say sort of LRU.

So, sort of LRU basically says that let us say instead of putting a full-time stamp on your page, on your axis let us approximate the time stamp by some number of bits alright. So, let say I have a 1-bit timestamp. So, instead of a timestamp that is likely to be you know very long timestamp you know I just have a one-bit timestamp and that timestamp is going to say whether this page has been accessed recently or not right.

So, each let us so, basically the idea is that there is just 1 bit inside this page table entry. If this bit is and this bit is call the access bit which says whether this page has been accessed or not right. When you load the page initially you clear this bit, you set it to 0 and then when it gets accessed the hardware will set this bit to 1. If this bit is already 1, then it gets accessed, then it remains 1 it does not change right.

So, it is a 1-bit approximation to a to a full timestamp, basically saying whether it was accessed at all since a last time it was loaded or last time it was checked right. So, let see what a how this works.

(Refer Slide Time: 35:19)



So, this algorithm is also called the clock algorithm, which is 1 bit, now can be thought of as a 1-bit LRU ok. Each page has an accessed bit or a reference bit right; hardware sets this one use and OS periodically clears it alright. So, the idea is that the operating

system will periodically look through all its pages and the pages which have the accessed bit set it will clear those that bit.

And, then it will again periodically look for it and then if it finds that the accessed bit has been set between the last time it looked at it, then basically means that it was accessed in this region. So, it basically means it was accessed recently, something that was not accessed in this time period was not accessed recently. So, it is a one-bit approximation to LRU in the sense that you are differentiating between a page that was accessed recently whose access bit is 1 and a page that was not accessed recently whose access bit is 0.

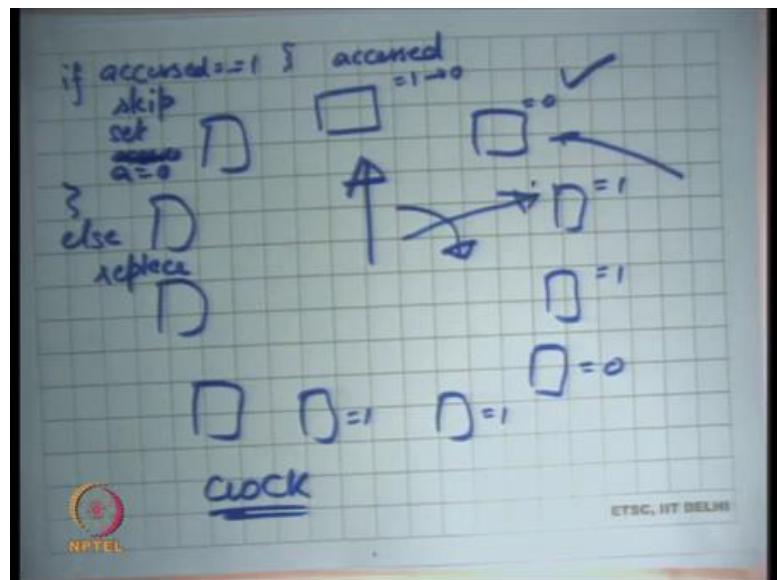
And, the notion of recently is basically the time between two checks by an operating system alright. So, instead of doing a full timestamp and sorting on the timestamp you just basically divide the pages into two categories accessed recently and not accessed recently.

And, you replace the page which has not accessed recently, and you do not replace the page that has been accessed recently alright ok. So, the way it works is that replace pages not accessed recently in FIFO order alright. So, FIFO had some merit to it right, FIFO basically said that whoever comes first goes first right.

And, it has some merit because it is quite likely that your programmed first is looking at this data structure and so, it loaded with lot of pages and now you are looking at this. So, FIFO does captures some part of recency in its (Refer Time: 37:41).

It is the recency of the first access to the page and the nice thing about FIFO was that it was very easy to implement right. LRU was much finer grained in figuring out who accessed what recently, but the problem with the LRU is that it is too expensive to implement. So, clock is a 1-bit approximation to LRU that does 1 bit of differentiation between what is accessed recently or what is not and, for everything else and so for pages in the same category it just use a FIFO alright.

(Refer Slide Time: 38:17)



So, you know purely implementation standpoint one way to think about the clock algorithm is that you arrange all the pages in a clock in a circle. And, your hand that is your clock hand that just moves let say clockwise alright. Let say you want to replace a page; you just move the hand till you find a page that has accessed bit equal to 0. So, each of these pages as an accessed bit equal to 1 0 1 0 1 and so on let say.

And so, the clock hand is going to so, let say I want to replace the page I will move the clock hand. Any page that I find that it has been its accessed bit is 1 I will skip it, I will not just skip it I will also change it to 0. Basically, means that I have seen it in this my in one revolution, I have seen it and I saw it has accessed. And so, I clear it to 0 to basically see that by the time I reach it next has it been converted to 1 or not.

The moment I see a page that has not been accessed since a last time I cleared it which means its accessed bit is still 0 I will take that for replacement right. So, this algorithm I mean this is this algorithm is basically doing what I said before its differentiating between pages that have been accessed recently. And, now here recently means accessed in the last clock in the last revolution of this clock hand, in the last one full revolution of this clock hand right.

So, its distinguishing page between pages that have been accessed recently and pages that have not been accessed recently for the pages that have not been accessed recently, it is replacing them in a FIFO order alright. So, basically why is it a FIFO order? Anytime

you pick a page to replace; so, let us say you pick this page to replace you will add the new page at this location and move the clock hand next right.

So, the page has just been the new page will get added just before the clock hand right. So, the pages the all the not accessed pages are replaced in FIFO order because, of that. The clock hand just sweeps through all the pages, the algorithm may if it finds the page that has bit equal to 1, it marks into 0 and skips it. If it finds a page that has bit equal to 0 it replaces it alright and it puts the new page at that position and the clock hand just points to the next page after that. So, it is pushed at the tale of the tube in some sense.

Student: And, the pages which are not sweep, they are not set to 0.

The pages that are not sweeped are not set to 0, but they will be set to 0 the next time the clock moves in that direction right. So, even the pages that have been set, that have been accessed are being examined in FIFO order right. There is there is an order in which they have basically being examined. So, we basically have two categories; recently accessed, not recently accessed and you are processing both of them in FIFO order.

And, one way to think about it is basically you arrange the pages in a clock, and you let the clock hand rotate. And, you use this algorithm that if accessed is equal to 1 skip set accessed is equal to 0, set a is equal to 0 let just say that alright else replace. So, each time you need to replace a page you just move the clock hand and you basically figure out which page to replace alright.

When you replace you would you know should you set a is equal to 1 or should you set a is equal to 0? Both are possibilities, but typically you would set a is equal to 0; we will let it remain as it is and see where the page program accessed or not.

Likely, to that program is going to access at after all you know that is the reason it being brought in if its it should be, but let say you are doing read ahead then you know you brought in some extra pages. And so, you have set the access bit to 0 and so, and if let say the read ahead pages were not the read ahead pages were not access.

So, then they will get replace in the next revolution alright. So, this is how it works. So, what does it mean to say that the hand is sweeping really fast? Now, it just going; so, each time you know by saying that the hand is sweeping really fast I am saying, if there

was a page fault and I wanted to bring a page in I had to go through a lot of pages before I could find something that could be replaced. What does that mean?

Student: (Refer Time: 42:52) the pages are used.

Student: Pages are being (Refer Time: 42:54).

That basically means that if the pages are really being used, many pages have been used in other words.

Student: (Refer Time: 42:59).

The hot memory footprint of the program is very large, it is probably larger than what the memory can actually support. In other words, you need to increase the size of a physical memory right. So, if the physical memory is smaller than the size of the hot memory footprint, also call the working set of the program then you have the you will see that the hand is has to move a lot of huge in a very fast right.

Let say I am doing a completely synchronous replacement, basically it may happen that each time I want to replace a page all the pages have been accessed before I get to I get page before I basically see a page.

So, I have to do a one full revolution before I actually get to do a replace of page. So, for every page that I replace I basically do one full revolution; so, that is you know that is really bad. You are basically doing a full global operation going through all the pages and also means that you are basically having a lot of you know you would likely to have a lot of misses.

If the miss rates are very high and shows that you are you need to increase your memory size. On the other hand, if the hand is sweeping really slow it basically means that your memory is larger than your working set size or the set size of the hot memory regions in your code.

Student: Sir.

So, the question is that.

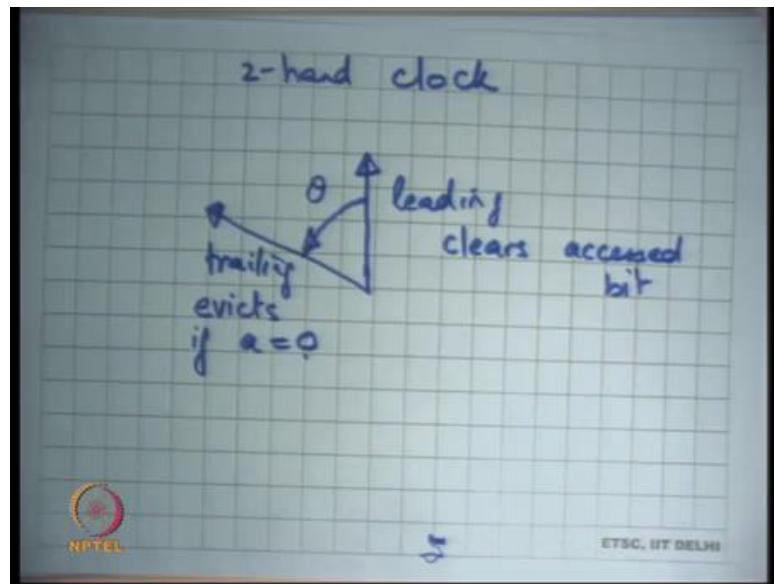
Student: So, all the (Refer Time: 44:13).

If I have to make a full revolution that basically means that all these pages were hits before I actually find something that I actually get to replace right. So, does not it mean that because all these pages were hits, does not that mean that the miss rate is small. Well, no not really right because all these pages are hits, but there are other pages that are also getting accessed basically means that all these pages are.

So, all the pages in your memory are hot and yet you have basically bringing in you know you are taking misses on other pages that also might be hot right. So, you basically taking misses and you your working set size seems to be bigger than your physical memory; so, you likely taking a lot of capacity misses.

It because all these pages are hot; so, you are finding it difficult which page to replace, that clearly shows that you know your miss rates are likely to be high ok. Another sort of modification to this is what is called a 2-hand clock.

(Refer Slide Time: 45:17)



So, if your memory size is really big then you do not want to make a full revolution on your entire memory on every page fault. And so, what is done is basically instead of having 1 hand, you have 2 hands. One is called the leading edge and other is called the trailing edge. The leading edge clears accessed bit and the trailing edge evicts, if accessed bit is equal to 0 right.

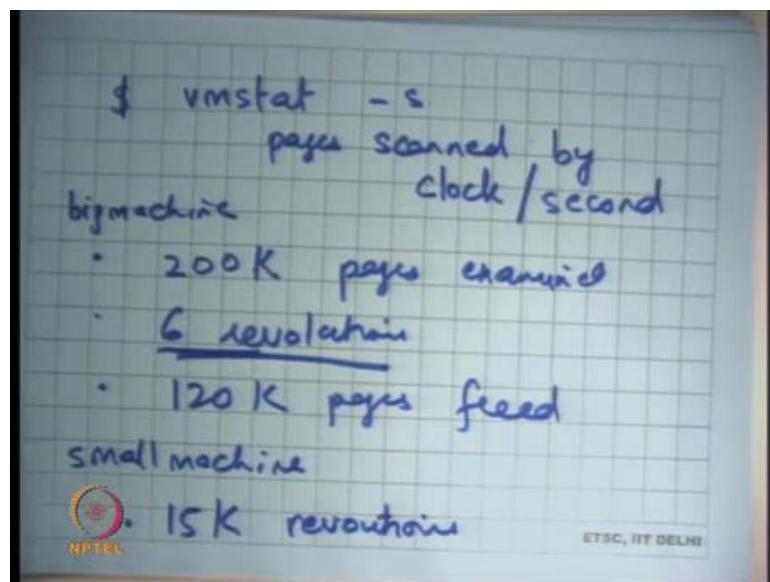
So, what have you done here? You are basically set that the leading edge clears the accessed bit and the trailing edge evicts, if access is equal to 0. If you are saying that the page will get evicted, if it gets accessed in this interval. In the 1 hand clock I was saying that the page will get evicted, if it was accessed in one revolution.

Here I am saying the page will get evicted, if it gets accessed in this interval and both of these; so, this angle is fixed. Let say this angle is theta. So, the theta is fixed. So, whenever that leading edge moves the trailing edge moves along with it right and so, a page get replaced if it was not accessed within this theta interval.

So, instead of theta being 360 as it was in the case of 1 hand clock, now theta is something smaller and you can choose theta depending on what your memory size is and what is the maximum overhead you want to have on your page fault. So, and so this basically make sure that you know you will examine only a that many number of pages before you actually find something to replace alright. If the angle is too small what does it mean? Let say theta is equal to 0, it basically means it is a FIFO because, the leading edge clears the accessed bit and trailing edge is just evicts it immediately.

So, basically means you did not care about the accessed bit at alright; on the other hand, if theta is equal to 360 you are back to 1 hand clock alright.

(Refer Slide Time: 47:37)



So, you know just as an example let say you know, I had a machine may find you know if you there is a command called vm stat on Sun OS. And, which allows you to check how many pages were scanned by the clock algorithm on operating system. So, clock algorithm is a commonly used algorithm in inside operating systems.

And, you can check what are the statistics of your clock algorithm on your operating system which will be using this command on some operating system, that tells you what are the pages scanned by the clock per second. So, for example, you know here is some example data. So, on a big machine big machine means lots of memory this slightly whole data, but let say you know roughly 200,000 page is examined. 6 revolutions of clock hand and 120 K pages freed alright, roughly speaking.

So, you know the 6 revolution basically shows that the clock hand is moving relatively moderately, its memory pressure is not that much. On the other hand, if you have a small machine you know here is another piece of data you could have something like let say 15,000 revolutions per second.

This basically shows that your memory is basically too small and you basically need to increase your memory to reduce your memory pressure and so, that your clock hand moves slowly alright.

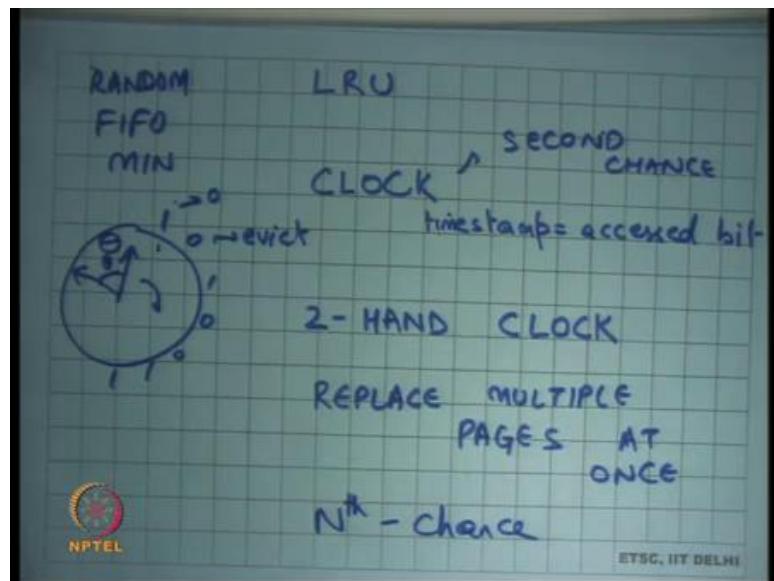
So, let us stop here and continue this discussion next time.

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

Lecture - 30
Page Replacement, Thrashing

Welcome to Operating Systems, lecture 30, right.

(Refer Slide Time: 00:31)



So, last time we were talking about Page Replacement algorithms and we looked at random, we looked at RANDOM, we looked at FIFO. We also looked at a hypothetical algorithm which we called MIN which is let us say the optimal, which assumed the knowledge of future and we said that assuming that past is equal to future LRU is the best approximation of MIN. So, if past approximates the future LRU approximates MIN.

But then we said LRU is too difficult to implement because you need to put a time stamp on every memory access and recall that memory accesses are on the hit path and they need to execute on you know very fast. So, putting a timestamp on every memory access is a very expensive operation. So, it is not practical in general.

So, LRU, so implementing perfect LRU is hard because you need to put a timestamp on every memory access and that needs to be on the hit path. And the you know, so putting a timestamp on every memory access is not possible instead clock is a 1-bit

approximation to LRU where the timestamp is just 1 bit which says whether it was accessed or not. So, there is just 1 bit and this bit is stored in the page table entry. Recall that there were these extra bits in the page table entry and so one of them is used on the (Refer Time: 01:52) disk hardware to represent the access bit.

And so, the hardware basically sets the access bit. The first time the page if the access bit was 0 and the page gets access then the access becomes bit becomes 1. If the access bit is already one and the page gets access nothing happens, right it just remains 1. So, that is a 1-bit approximation to LRU, and we said that you know that basically. So, we basically use FIFO except that we use this 1 bit to distinguish between pages that were access recently and that were not access recently.

Pages that were accessed recently are not replaced they are instead move to the other list for the first time. So, you basically examine these pages in FIFO order and if you see a page that has been accessed recently you clear its access bit indicating that it has been moved to the lower level, but if you see a page in the lower level in FIFO order then you basically evict it. The first page you see it, right.

So, that you know the one another way to think about it is that it is a clock and you keep a pointer to the last page that you accessed and you move the clock in one direction and these clocks have you know access bits 0, something like this. And, each time you come across a page whose access bit is set you set it to 0, and if you see something which is already 0 you evict it, right. And because you are storing the pointer, the next time you run the page replacement algorithm you going to start from the where you left off. So, it turns it looks like FIFO in that sense, right. Basically, you are just advancing the pointer from the from where it was last time.

What is the logic behind setting the one, setting the access bit from 1 to 0? Well, I mean basically the idea is that if a page was is 1, then I set it to 0, so that I want to observe whether it is you know I have seen it at once and now the next time I am more interested in seeing whether it was access in the last revolution or not, right. I do not want to carry over information from 10 revolutions behind, I am not interested. I am only interested in knowing whether it was accessed in the last revolution or not and to be able to do that I need to clear it. So, that it gets set again.

Student: So, cannot we traverse set all of them, set all 1s to 0?

Cannot we traverse all of them and set all of them to from 1 to 0? I mean this is this is in some sense doing that except that it is not a global operation, you are just going in a sequential way. I mean the net effect is similar, right. You are just going one by one and whichever you see as 1 you send it to the 0, and so each page is getting in equal time in to get accessed which is one revolution or roughly equal time you know on average. It is also metric anyways, right.

And, and, but of course, we said that you know if the number of pages is really large and the time that a page gets to prove that its actually being accessed recently maybe too large for your for what you need and so you may want to use a 2-hand clock. I am just say that the leading hand is going to clear the bit and the trailing hand is going to evict. So, the leading hand just clears the bits if it finds the 1 and the trailing hand evicts if it finds a 0. And so, for a page to not get evicted the page should have been accessed between the leading hand and the trailing hand, that is the only way the page will survive eviction. It will not be able to survive it is the past.

And so, you can tune that by using and so this angle is fixed between the leading edge and the trailing edge that is let us say, let us call that theta. So, this angle is fixed and depending on how much time you want to give to a page to really allow it to be called accessed recently you will choose theta. If you choose theta to be too large you basically giving it the page more time, if you choose theta to be too small then you are giving page too less time. And you know extreme cases are theta is equal to 0 you are actually not giving the page any time at all and it is just FIFO at that point. And if you make theta is equal to 360 then you are basically back to 1-hand clock, right, alright.

So, there are some, so you know the clock algorithm is fairly successful in the sense that it is actually being used for many years to do page replacement and here are some more extensions to the clock algorithm that are usually used.

So, one is you know replace multiple pages at once. So, instead of replacing one page at a time; so, each time you get a page fault you may you actually just obliged to replace one page instead of replacing one page you may want to replace multiple pages at once; so, that you would save you know. So, firstly, if there are any 30 pages you can write all of them in a batch to the disk. And, so as we know that disk is dominated by the seek and

latency time, rotational latency it is better to write a bunch of pages together rather than one page at a time. Also, it saves; so, yeah, so I think it is basically about you know saving the number of writes that you can have.

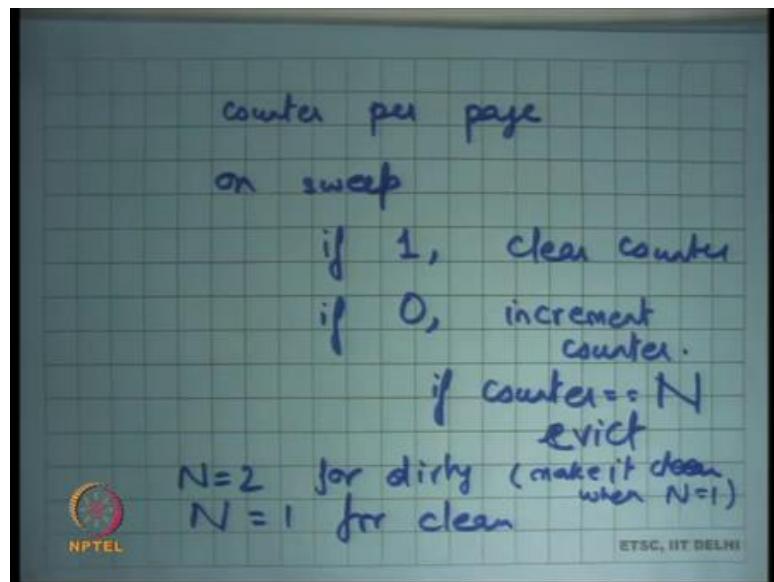
Another sort of straightforward extension to the clock algorithm is what is called the Nth chance algorithm. So, one way to think about the clock algorithm is it is also called a second chance algorithm where pages that have been accessed recently are given a second chance, right.

So, you basically process all your pages in a FIFO order, but if you see that a page has been accessed recently you give it a second chance which means you set it to 0 and you give it another chance to basically prove itself once again, right. And, if it is able to prove it is again which means it gets access recently you given give yet another chance, but if it is not able to get access recently then it gets evicted. So, it is a second chance algorithm.

The Nth chance algorithm is a just a straightforward extension you say instead of giving it two chances give it N chances. So, the idea is that each time you see, so you go along the clock you see one bit, you set the counter, so we maintain a counter with every page which says how many times you have seen this page to be not accessed, right; so, how many rounds have you seen this page to be not accessed

And each time you see a 1 bit in that in that page you clear the counter to 0. Basically, means it was just accessed recently to the counter 0. If you say see it as 0 then you increment the counter, you do not replace it immediately you increment the counter by 1, right. And when the counter reaches N which is where N is the you know Nth chance of then you replace it, right. So, let me just write the Nth chance algorithm to make it clearer.

(Refer Slide Time: 08:29)



Counter per page on sweep by the clock hand, if you see a 1 then clear counter, if you see a 0 then increment N, increment counter, right. Also, if counter equals 1 replace or evict, alright. So, in other words the clock algorithm as we seen at the second chance algorithm is just Nth chance with N is equal to 1. The first time you see it as a 0, you increment the counter and the counter becomes equal to 1 and you replace it. Did I say N 1, it should be N.

So, the clock algorithm is the Nth chance with N equal to 1, but if you want to give a page more chances to basically prove itself then you know you increment. The more, the higher you have the larger N, the Nth the more revolutions you have to make before you actually evict a page or to actually see the page as not accept for at least you know N revolutions before you actually evicted. So, giving a page more chances in that sense; also, the higher the N the closer you are to LRU, right.

So, basically let us say N was equal to 1000 basically saying that this page has not been accessed the 1000 times and so this is the best candidate to get evicted, right. On the other hand, if there is a page that was accessed in the last 1000 revolutions before you know will get priority over, so you are getting finer grained. You are distinguishing between a page that was accessed a 1000 times and a page that was accessed that was not accessed across a 1000 revolutions and that was not the page that was not accessed across 900 revolutions let us say, right.

So, the higher the N the more fine grained you are getting. You are distinguishing between pages that have been not been accessed a 1000 time for a 1000 revolutions and a page that has not been accessed for a 900 revolutions. So, the higher the N the closer you get into LRU. The downside is the more work you have to do to find a page to replace, right, ok, alright.

So, you know if this is, so the Nth chance algorithm with a large N is one way to approximate LRU, but in general clock algorithm works just as well as LRU. LRU is anyways an approximation to the what is optimal and clock algorithm is approximation to LRU and in general it works good enough. So, you do not need to do these Nth chance or kind of things or you do not need to have a very large N to basically have good hit ratios, right. The whole point of doing all this is basically to have good hit ratios without having too much overhead on your hit path and also on the miss path.

One common thing that is done is treat dirty pages preferentially over clean pages. So, when you are doing replacement you can imagine that if there if you make a dirty page you will have to do more work because dirty page if eviction requires a write to the disk, a clean page eviction does not require a write to the disk. So, keeping this in mind you may want to give more priority to your dirty pages. So, you may say what is the dirty page let me give it more priority over your clean page. So, if I have a choice between dirty and a clean page, I will probably pick the clean page to evict, alright.

So, you know one common way that is sometimes done is use user Nth chance algorithm with $N = 2$ for dirty pages and $N = 1$ for clean pages, right. So, evict the clean page the first time you see it not accessed evict the dirty page the second time you see it not accessed, alright.

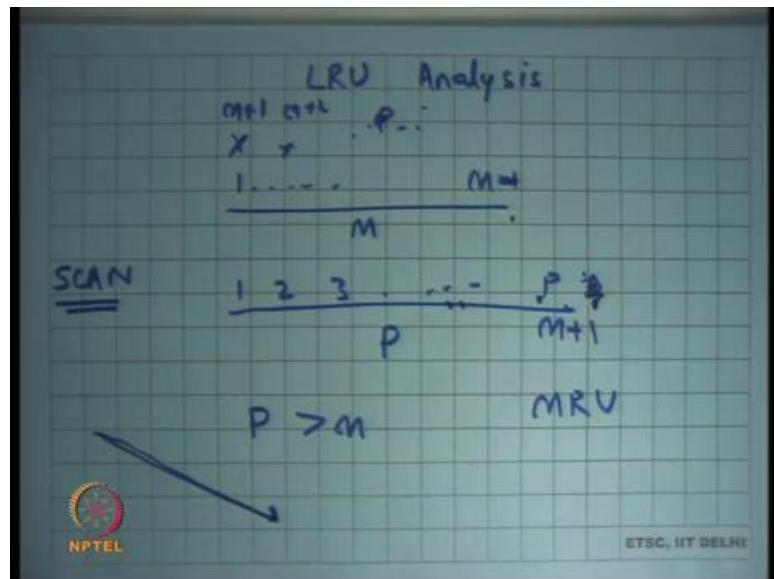
Also, in this scenario you make it clean when $N = 1$, right. So, let us say you went through that page and you figured out that it has not been access in the last revolution. So, you do not evict it immediately, but you also say that you know let us just write it back disk, so that by the time I come to it next it becomes clean.

And you know this, right back to the disk can be done in an asynchronous fashion in a batch fashion, so lots of pages, lots of dirty pages can be collected together and written

back. So, if a page has not been seen since access in the last revolution which is you know at $N = 1$ for the dirty page, you can just put it in your queue to be written back to the disk and you know because it is likely to get replaced in future, there is some indication to use there. So, you make it clean at $N = 1$ and you replace it at $N = 2$, right. For clean pages you can just replace it at $N = 1$.

So, just one example of Nth chance of algorithm used and to potentially treat dirty pages over clean pages, alright. So, let us just understand where LRU works and where LRU does not work, right.

(Refer Slide Time: 13:39)



So, LRU works very well basically LRU captures the recency how recently you have accessed the page and in general our workloads are you know have a lot of similarity between past and future and so recency works.

But here is an example; here is a common example of a pattern where LRU does not work. So, let us say you have a, you have some physical memory M and you are accessing a set of pages P such that $P > M$. Now, let us say you are going to say accessing 1, 2, 3 P and you were just accessing these pages in let us say repeatedly in a cyclic manner 1, 2, 3, 4, 5, 6, till P and then you come back to 1 and so on, right.

Now, in this case let us see what happens with LRU. So, you accessed 1, 2, 3, 4, 5, 6, till N , all of them are misses in the first iteration and so, but all of them get into M . So,

your N basically becomes 1 M - 1, 1 M. Then you access M + 1 and which is the page that you replace?

Student: 1.

1, right. So, you replace 1 and you put M + 1 here and you replace 2 and put M + 2 here and so on, right, till let us say till let us say P, hope you have some. And then you access 1 again. The problem is that you know 1 the page that you replaced is also the page that you are going to access. So, here is a case where past is not equal to future. The page that has access least recently is the page that will be accessed most closely in future, right.

In this sense, in this case instead of replacing 1 it would have been in fact better to replace the page M because M is the page that is going to be accessed latest in future, right. So, so basically you know instead of LRU perhaps MRU most recently used which is you know completely counter intuitive would have worked in this in this example workload. Just pick the page that was most recently used that is a page that is you know going to be accessed further in the future and so that is going to have some hit rate. LRU in this workload will have completely 0 hit rate, you know all the pages will be all the accesses will be misses.

And this kind of a pattern is actually quite common for example, you know process going through an array of just scanning its array linearly or going through its data structure like a tree or whatever. So, basically any linear scan and repeated is quite a common scenario in real workloads and that is why we have to worry about it ok. So, what happens with a scan?

So, this kind of workload is also called a scan and notice that the worst performance of the LRU seems to perform the worst if P was equal to just $1 > M$. So, let us say $P = M + 1$, so basically for the M + 1st page you replace the first page whereas, the next page you want to access this 1. So, you basically replacing the page that is most likely to be accessed next, right.

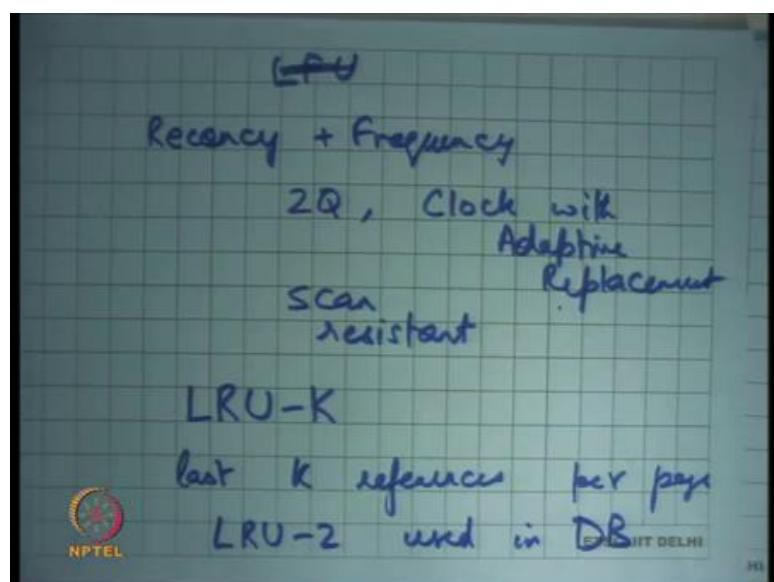
And so this kind of pattern is called a scan. And the problem with scan is that let us say typically your operating system will have multiple processes running or multiple threads running or at least multiple access patterns. So, there is somebody who is accessing it in a linear scan and there are other parts of the program that are just accessing it in a very

differential way; that means, they are always accessing the same locations again and again, but one scan can actually pollute the cash by bringing in lots of code pages into your cash.

So, all your hot pages get flushed out and the cold pages because of the scan get place in your cache and so that causes a lot of misses. Ideally, it would have been nice if your algorithm could figure out that hey this is a scan and so I do not need to do any caching for these pages, right. Anyways these are going to be misses, and rather I should focus on doing caching, I should focus on retaining my hot pages. These, all these P pages are anyways cold pages, so let them have misses. These hot pages should not get polluted because of the cold pages.

And the reason is this problem is occurring is because LRU was only looking at recency, it is only looking at what was the least recently used throwing it out, it is not looking at frequency at all. So, if there are some hot page it is getting access to 100 times and there is a cold page that was just accessed once recently, the page the cold page gets preference over with the hot page just because it was access more recently. So, ideally it should we should have some way of figuring out, some way of combining frequency also with recency, alright. So, you know there is there is an algorithm called least frequently used.

(Refer Slide Time: 18:29)



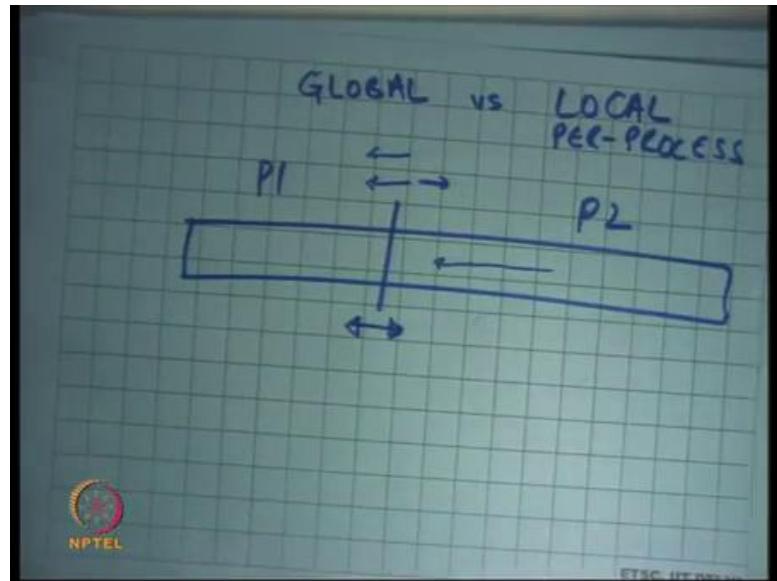
But this is also very impractical algorithm, you basically say what is the least frequently used page, but the problem with this is that it has a long memory. You know a page that was accessed a lot of times one hour back may just keep polluting your cache which is not known typically your work typical workloads will perform badly in LFU.

So, typically you combine recency and frequency in some way and so LRU is not used in explained form and there are examples of algorithms which extend either LRU. So, there is an algorithm called 2Q, there is an algorithm called clock with adaptive replacement and so on which basically does and there are many others. So, most operating systems will implement some variant of LRU that is scan resistant, right. So, because scans are common workload patterns and any good algorithm should also be scan resistant and LRU is not, ok, alright.

So, one algorithm that I will just give you a flavor is called LRU-K. Here the idea is that you do not evict the page that was least recently used, you for each page you track the last K references, alright and then you evict the page which has the least timestamp for the Kth reference in the past.

So, in LRU we were just looking at the last reference for the last accesses, instead of the last timestamp with the last access you look for the last time stamp for the last Kth access and that is the one new replace. So, you know LRU-K is basically giving some preference to frequency and, but the LRU-K is also expensive to implement in the context of an operating system, but LRU-2 is used in databases, alright. Just, ok, alright.

(Refer Slide Time: 20:55)



So, now let us talk about how something like a cache replace in algorithm is implemented and there are two choices global versus local or per process. Should I do replacement for all the processes as considered as one pool or should I do replacement for each process separately? Right. There are two options I can consider the memory used by all the processes as one global pool and so in one page process page faults I will consider this whole pool as one replacement pool and I can take page from some other process to serve this process and other cases I have separate pools for process, right.

So, so clearly if you use global then you have more optimality overall. If you have per process quotas of memory then it is possible that one process is under utilizing its memory and another process is over committing its memory and so you know you have artificial barriers and that is causing extra page faults and you actually need to. On the other hand, global replacement has its problem that one process can actually run away with all the memory causing all of the process to become perform very poorly, right.

So, a typical security attack or you know performance attack on a machine could be that you respond a process and it just touches a lot of memory, right. And, just keeps touching a lot of memory and so it just bringing in a lot of memories all other processes is much less memory than actually available on the system, right. And so, you know the operating system has to balance optimality or efficiency with some level of fearness ok.

So, in practice you basically use, so in practice there are two things that are typically used, one is you know either they use completely global replacement. So, you know just global replacement assuming you have a very large memory pool and your caching parameters are correct; you know it is very hard for one process to actually pollute the cache of other processes.

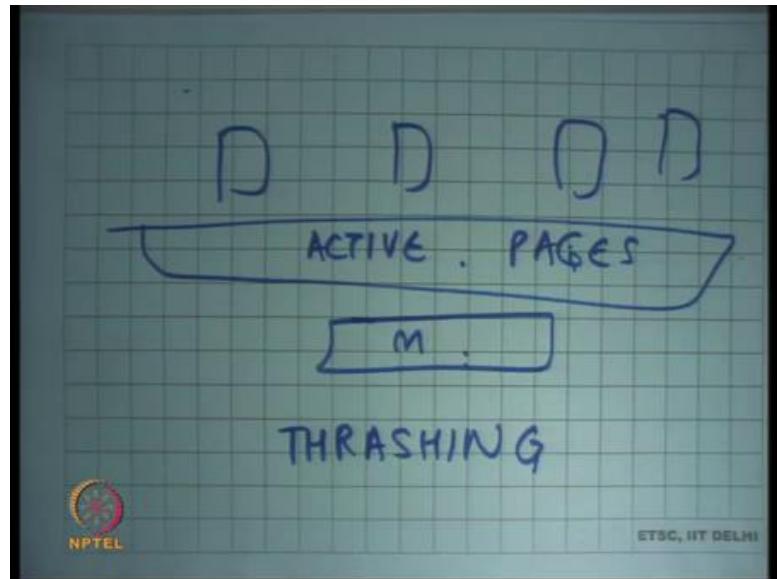
The other way to do it is basically have per process pools. So, you have per process quotas, let us say you know I have a quota for let us say this is my complete memory and I have some quota for process P1 and I have another quota for process P2. And then I basically if process P2 is taking lots of page faults then slowly I will move this barrier in one direction or another.

So, if I find out that P2 is taking lots of page faults and P1 is under utilizing its memory, then slowly maybe on the order of minutes I will basically figure out and not try to move this quota allocation one direction or the other. The important note there is that the movement of this allocation boundary is much slower than the boundary, than the actual page replacement rate, right. So, that gives you both some level of fairness and yet not too much inefficiency, right. So, it gives you fairness because P2 gets, P1 gets isolated from P2 to some extent.

At the same time if the operating system figures out that P2 actually needs a lot of memory and P1 does not need that much memory then there is some adjustment at a time level granularity to make to account for that, alright, ok, good. So, this is you know so we have discussed demand paging and we have said that you know pages are brought as in when they are required and this page replacement algorithm going on.

And in general if a process takes a page fault it has to wait for a disk request and while the disk is actually serving this process you will probably run another process and so you know the process that the cost of page faults actually gets hidden and the page faults in some sense become free, right. But, let us see what happens if it memory gets overcommitted, right. So, let us say the sum of all the memory that is required or all the active memory that required all these processes is greater than the physical memory that you actually have, right.

(Refer Slide Time: 25:07)



So, let us say you know there are processes running, and you have physical memory, but you know the total amount of memory is smaller than the amount of physical memory that you have. This is greater than this. So, this is let us say the active set, active pages of all these processes and this is the physical memory, and this is greater, alright.

So, what is likely to happen is processes are likely to take lots of page faults. So, let us say this process takes a page fault it is likely that it will go to the disk and in and it will replace one of these pages and these pages, and so one of the pages that are actually resident has going to be replaced and because all these pages are active most likely you will replace an active page.

And so, because you replace an active page very soon you will get another page fault because of that replacement, and so eventually what will happen is all these processes will just keep taking page faults, right. So, one-page process takes page faults and the other processes page, that process takes a page faults it replaces this processes page or its own pages and so you are basically spending a lot of time on page faults.

And so, what is happening at this point is that the system is spending a lot of time taking page faults and reading and writing pages from the disk and not actually executing useful instructions. The program was written to execute useful instructions, but these instructions are just page faulting and the and for no fault of theirs, it is actually the operating system that was trying to play tricks under the carpet and providing a very

fancy revision of a large address space with x performance latency of a of physical memory.

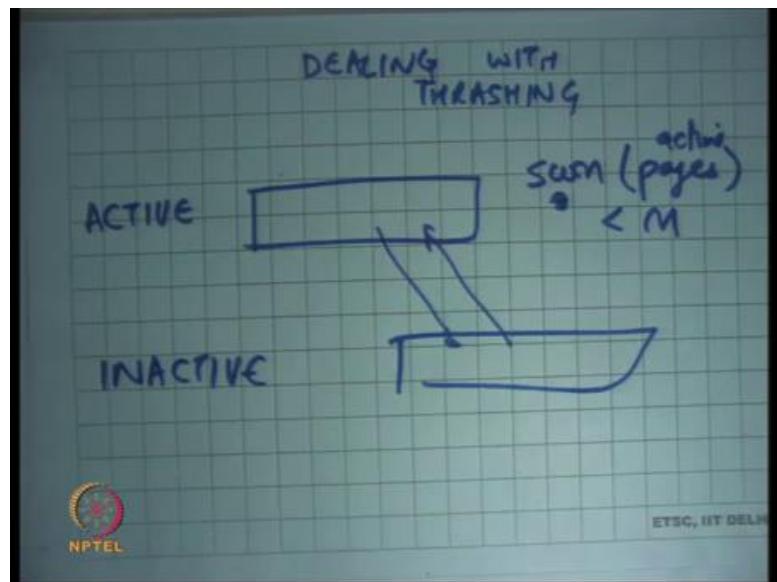
But what is what a at this point what has happened is you have an address space, alright, but the access latency is not a physical memory, the access latency now becomes of the disk, right. So, each memory access now becomes disk latency access at this point and so this situation is called thrashing, right. So, at this point the operating system is thrashing or your computer system is actually thrashing which means most of the time is actually being spent on reading and writing data to and from the disk and not actually executing useful instructions.

And you may have seen this in real life, in practice if you spawn too many processes you may have seen that is the sometimes the computer becomes really slow depending on you know if you are if your memory is not big enough to handle that many processes. And you know your let us say your flickering light for your hard disk access is continuously flickering which means the hard disk is continuously being read and written to, but the computer system as a whole seems to be very slow and really at that point your system is thrashing, right ok. So, and so typically what do you do to resolve the situation in practice?

Student: (Refer Time: 27:23).

You close some processes, right. And so, why do you expect that if you close some processes things become better? Because you close some processes then that you know they will reduce the set of active pages. So, some processes are not active, so their pages have been removed from the process and now the set of active pages fits in the physical memory and so at this point you can now start executing from physical at physical memory speed back again, alright, ok. So, an operating system automatically also can try at least to do the same thing. So, how does an operating system deal with thrashing?

(Refer Slide Time: 28:33)



So, let us say dealing with thrashing. Well, if just one process is causing all the thrashing then there is nothing the operating system can do about it. That one process is accessing so much memory or so many active pages that it cannot fit in the physical memory and there is nothing the operating system can do about it. The maximum it can do is you know schedule that process less often, so that other processes are not affected or maybe even kill that process, right. So, those are the only two options just one processes causing all the problem.

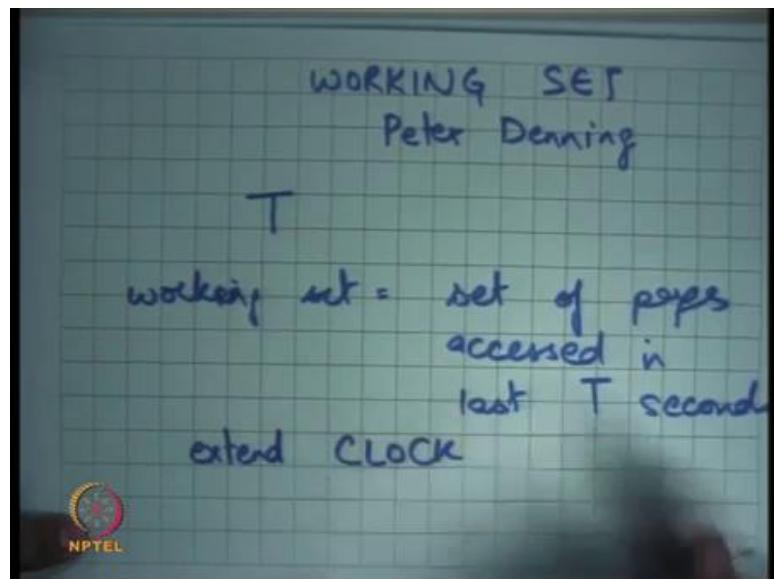
The more common case is that the sum total of all the running processes is causing all the problems, and so in that case the operating system can do something like that. It can figure out which processes using how much memory and then schedule the processes in groups. So, have two groups, one is active processes and the other inactive processes. And so at a time you will have only some set of processes in the active processes and make sure that the sum of all the memory active memory usage of those processes is smaller or comfortably smaller than the size of the physical memory, right.

So, basically you know have set of active processes and have a set of inactive processes and keep moving processes from active to inactive, right. And make sure that sum of sum of pages active pages inside of the active processes is less than physical memory or comfortably less than physical memory.

The scheduler only schedules the active processes at a time. The inactive processes remain in the sleeping or suspended state or even in the ready state they are not actually brought up and started to run. So, it is a scheduler that is playing you know playing in consonance with your virtual memory subsystem to figure out you know which processes should be run at this point, so that you will the system does not start thrashing at this point ok.

So, the question is how do I figure out what is the size of the active pages of a process, right. So, notice I am not just saying that how much memory is allocated in the process that is not important. What is important is what is a set of active pages in a process which pages that have been being used by this process on a in an active way, right.

(Refer Slide Time: 30:59)



So, there is a way, so there is a term called a working set, used to describe the active set of a process and this was this term was or this method was proposed by Peter Denning, alright. And informally, the working set of a process is a collection of pages that have been that are actively being used by the process, alright, ok. And let us see how the working set is computed.

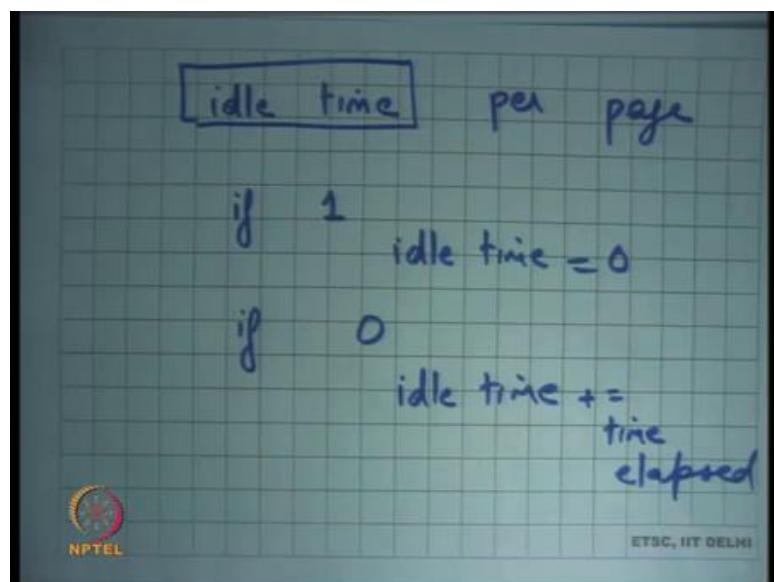
So, firstly, if you can compute the working set of every process then it is very easy, I basically you know make sure that the sum of working sets of the active processes is less than physical memory and the scheduler basically does this does this intelligently, ok. Also notice that the working set of processes may change over time. So, right now

the process is accessing let us say 5 MB of, has a working set of 5 MB, sometime later let us say a few minutes later it becomes 5 KB. So, I should figure that out I should and I should account for that in my scheduled algorithm, right.

So, now, let us first say see how a working set can be computed, alright. So, the working set is based on a parameter called T and working set is basically set of pages accessed in last T seconds, alright. So, if I define the working set as a set of pages that have been in the last T seconds, you know that is the one way of defining the set of active pages or process.

Notice that here again, I am using recency as a criteria to distinguish between active and non-active, right. Just like in LRU, I was using recency to distinguish between something that needs to be replaced or not. I am not using frequency at all and that is also primarily because workloads typically behave in a past is equal future manner and so recency is more accurate, guess then frequency, right. So, how do I compute the set of pages that have been accessed in the last T seconds? Well, one way to do this is basically extend the clock algorithm, alright.

(Refer Slide Time: 33:19)



With each page keep a field called idle time per page, ok. And in the clock algorithm, if you see a one then set idle time to 0 of that page, alright and if you see a 0 then increment the idle time with the time elapsed since the last time you saw it, ok. And so that way you can keep you keep track of in a proximate manner not completely precise,

but in an approximate manner what how long this page has been idle, right. And once you have an idle time per page your working set is basically all the pages whose idle time is.

Student: Less.

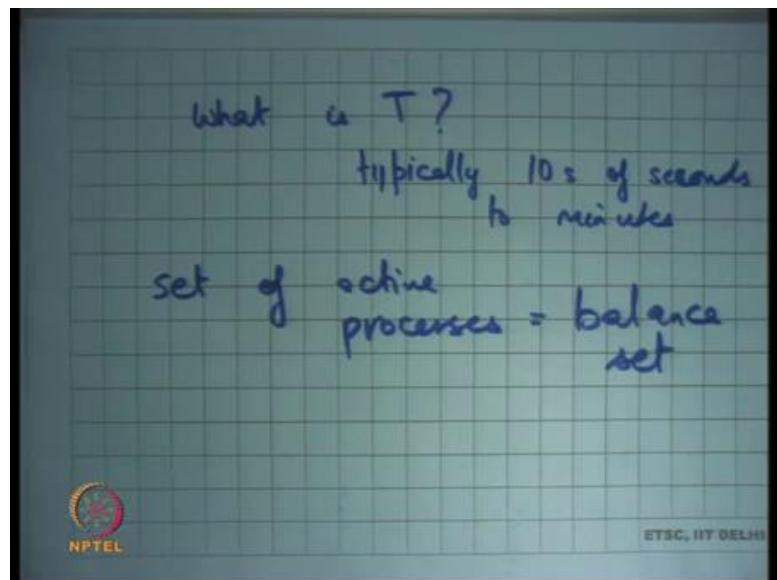
Is less than T, ok. So, you can compute the working set in an efficient way, and you can even use it. Yes, question.

Student: Sir, did I have to keep a record of the time elapsed for each of the page in each revolution?

So, keeping the time recording the time elapsed is you know you can have another field for example, idle time and last time it was set, right. And so you just have to make one computation to figure out you know how much time has elapsed, you know the current time, you know the last time you set it and you basically know how much time it is been set or you can use some approximation of this. In any case, recall that this movement of the clock hand is happening on the miss path, right and so this extra computation is not a big deal because you know it is dominated by the page fault handler and the disk access and all that, this extra computation is actually not too expensive.

In this competition of the working set no way have we increased the time or increase the overhead on the hit path that is the most important thing, right, ok. So, you know, so that is the working set and using the working set you can decide which processes to keep in the active set and that way you can prevent thrashing, alright. Some difficult questions for the working set idea. Well, firstly, how long how big should T be?

(Refer Slide Time: 35:31)



What is T? Right. Well, typically 10s of seconds to minutes, right. So, you use a relatively coarse grained value of T to figure out what are the set of active pages, you being a little conservative in the sense and set of active pages you being a little conservative to make sure that your system is actually completely divide of trashing. So, you have completely preventing trashing by having a large parameter for T, alright

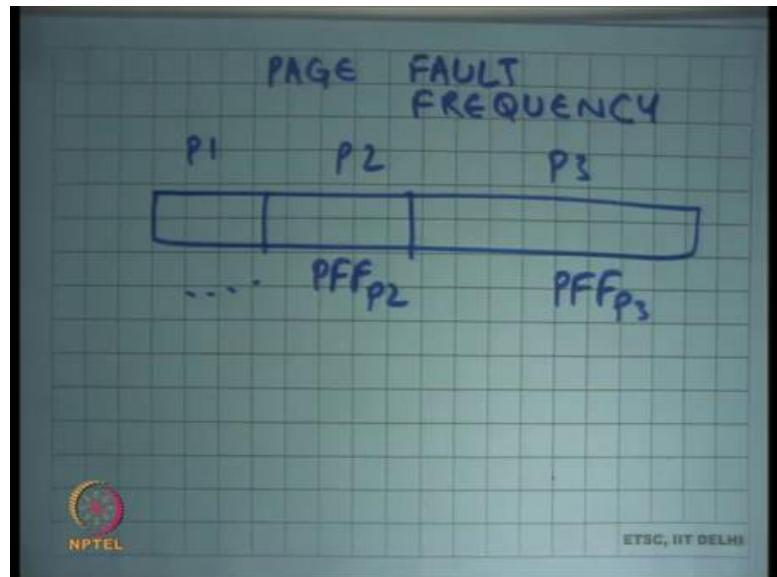
Also, you need to change handle changes in your working set. So, as the working set is changing you need to figure that out on the on dynamically and based on that change the set of active processes. Also, know this set of active processes is called the balance set balance set and so the operating system makes sure that the sum total of the working sets of the balance set of processes is less than the precise of physical memory, it is just a term called balance set, alright.

And, also you know when you are doing this kind of thing you have to worry about what if processes share memory. So, let us say there are two processes that share memory then you need to worry about that as well, right. For example, one thing to do maybe that these processes are considered as one unit, so they either together go into the balance set or they together go out of the balance set, right.

Because otherwise you know you have to account for, otherwise its more inefficient because shared pages are likely to be; so, it is better to account for shared pages you may want to do that if this number of shared page is very large if the number

of shared page is not very large you may want to have other ways to figure out you know what the working sets are ok, alright.

(Refer Slide Time: 37:47)



Another way of preventing thrashing is using the page fault frequency. So, one way of preventing thrashing was using the working set approach another way of preventing thrashing is look at the page fault frequency of every process. Here, basically you know you have let us say this is your physical memory you have per process allocations of physical memory, alright and you basically maintain the page fault frequency of each process.

So, let us say this is the page fault frequency of process P3 and PFF of P2 and so on, right. And if you figure out that one process has an extremely high page fault frequency and another process has an extremely low page fault frequency then you can adjust these boundaries based on that, right. So, as opposed to the working set which has which also has a tunable parameter called T, page fault frequency will also have some tunable parameters saying you know what is the threshold where you are going to move things and the.

Also, if you figure out that the sum of all the memory pools, does not fit in your main memory then you can deactivate some processes ok, alright, ok. So, this is an alternate idea to working sets to be able to manage your virtual memory and to prevent thrashing.

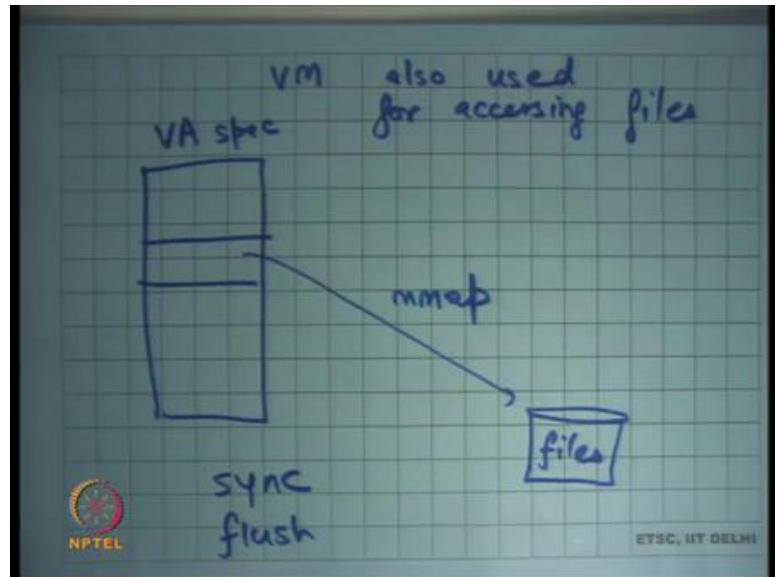
In general, thrashing was more of a problem. So, this thrashing was a big problem in the early days of shared computers. So, you know let us say in the late 70s or 80s when we are used to have large shared computers like mainframes and lots of people accessing these mainframes, then at that point you know you had to worry about thrashing and these kind of things to basically make sure that all the users get some level of fairness and also the maximum amount of performance that you can get out of a shared computer.

With the advent of personal computers this became less of a problem. Number 1, you could you know memory becomes became cheaper and so if you see thrashing the best way one way to do it is basically you will just add more memory and so you get rid of much of this much of these problems and the does not need to worry about it as much as it had to worry about it in shared computers. So, memory has become cheaper.

The other thing is the user has more control over a system. So, the user himself can manually by hand just switch off processes, but on a shared computer no nobody wants to stop his process because of the other process, right, but when you have a personal computer you know the user is doing more of this management itself. So, in other words basically you know in a personal computer the balance set it can be managed by hand by the user, alright, ok, good.

So, that is that is all I have for the virtual memory subsystem. And, next time I will start discussing about storage devices and what are the different kinds of storage devices we have. And, then and the kind of data structures we use on storage devices to implement abstractions like file systems, ok.

(Refer Slide Time: 41:09)



One last thing I would like to say is that the virtual memory subsystem is also used for accessing files in modern operating systems, right. So, how? Well, in your virtual address space you can; so, let us say this is the VA space, you can map regions of a VA space to point to files, right. And so, for example, on UNIX you can do this using the `mmap` system call, right.

And once you do that the program the programmer can now just treat this region as any other memory and do pointer dereferences on it, right. So, read and write can be done directly here and under the covers the operating system is managing reads and writes to the disk, right. Notice that this is an alternate way of accessing files from the one that we have discussed so far. So far, we are discussing ways of open, read, write, close, right.

Here I have mapped the file into my address space and then just use pointers or just use my program how just consider those start space like any other memory region, right. For example, I could manage a tree data structure in this space, right. And so, my program does not need to worry about that it is a disk that is that is that is getting written to at the back end.

To make it efficient of course, the operating system is caching pages in physical memory, right. So, it is not like every pointer dereferences going all the way to the disk, they are being handled by page faults and pages are being brought into physical memory so that most of your operations are fast, but they are also flushing demons and page

replacement demons are running at the background to basically make sure that the file contents are updated ok.

The only difference between using this virtual memory subsystem for other space and with using files is that for files the user always also needs some guarantees. So, for example, if I write something to this address space that has been mapped, what is my how do I know whether this has been written to disk or not, right. And I may need some guarantees over that. In case of virtual memory, I did not care about any guarantee I did not care if the thing was written to swap or not because when the system reboots the swap is anyways cleared, the swap spaces anyways cleared, but for the file I need some guarantees that it has to be written to the disk.

So, one way to do that is basically using you know providing system calls like you know flush or sync. So, you basically if the; in general when you write there is no guarantee that the contents will be reflected on disk, but if you want to have a guarantee then you can use an explicit system call to basically say that the contents should be flushed to disk and they will be flushed to disk in some dashed way, so that it becomes efficient also, right.

What are the advantages of doing file accesses using `m map` over doing file accesses using `read` or `write`? Well, one advantage I do not pay the cost of the system call. If I need to access the file, I just make a pointer dereference and that is it, right. I did not have to do a system call which involves a trap, some kernel code getting to run and so on. In this case, I can just dereference the pointer and assuming that the page is already mapped in my virtual address space, the operation will be logically done, not physically done, physically done later, but logically it is done.

Student: Sir. Sir, but if you just look it into, if we just read the file into a character array and then we perform all the computation on that array and later write it back to the disk. So, would it be the same thing?

If we read the file into a character array and write to the character array and then write it all back to the disk won't it be the same thing? Yes, I mean this is a way of the operating system providing you that functionality under the covers that you know there you have to do it explicitly. So, you have to read the whole file into your character array you. So, for example, if there are multiple processes who want to access the same file, so there is,

there needs to be some way of the file system is also giving you some sharing between multiple processes, but if you read into an a character array then there is no sharing.

For example, you know if there is a there is one file that is opened by multiple processes and so one process write to that file the other process can immediately read from that file because it is a shared address space, it is a shared file system, but file is the same. The obstruction is that it is being feared by multiple processes, but if you read a new character array then there is no sharing for example, right.

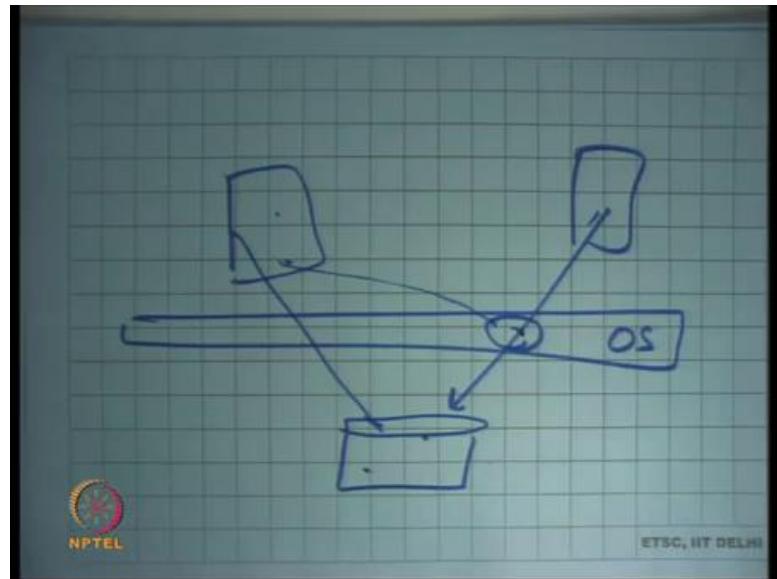
Also you know reading from a character array to into a character array requires a programmer, effort and programmer, understanding of what it means to you know when to write it back and optimization, it is better to rely on the operating system to do these things for you, right in a memory mapped environment.

So, you are saving the cost of a system call number one, but is that that important? Well, you know firstly, its if you are going to take a disk access then yes this cost of system call is not important, but if you are going to be if you were going to get served from the buffer cache then you know the cost of system saving the cost of system call is good.

The other thing is you know recently we have seen lots of different types of storage devices apart from magnetic disks that we have talked about there are things like flash memory and other sort of persistent memory, technologies, that are giving you very fast sort of persistent memory. They have their own problems, or you know quirks, but, but in any case, they you know they are providing alternate ways of providing secondary storage.

And, so accessing that if the latency of accessing that kind of secondary storage is very small then the cost of actually, doing the system call becomes large. So, let us say let us say hypothetically speaking that this was as fast as memory, right. In that case, using the read write system call to access the disk is very wasteful. You would want to access it with the same interface as you access memory, which is just pointed dereferences, right and so m map is a nice, elegant way of being able to do that.

(Refer Slide Time: 47:45)



So, we were going to discuss this in the file system discussion, but let us say this is the disk, the disk is the operating system sits on top of the disk, right. So, no process can touch the disk directly. So, if there is a process, if there are two processes that are accessing the disk the operating system is maintaining synchronization between them firstly. And so, if this one changes something it changes both in the buffer cache and in the disk, and so later if somebody reads read the latest value from the disk, ok.

So, the obstruction is that of a shared file. The operating system is responsible for doing synchronization which means that you know there should not be any race conditions on accessing access of the buffer cache, there should not be any race conditions on access to the disk, but at the same time the abstraction is that of a common file its not you know the obstruction is not of two independent files, ok.

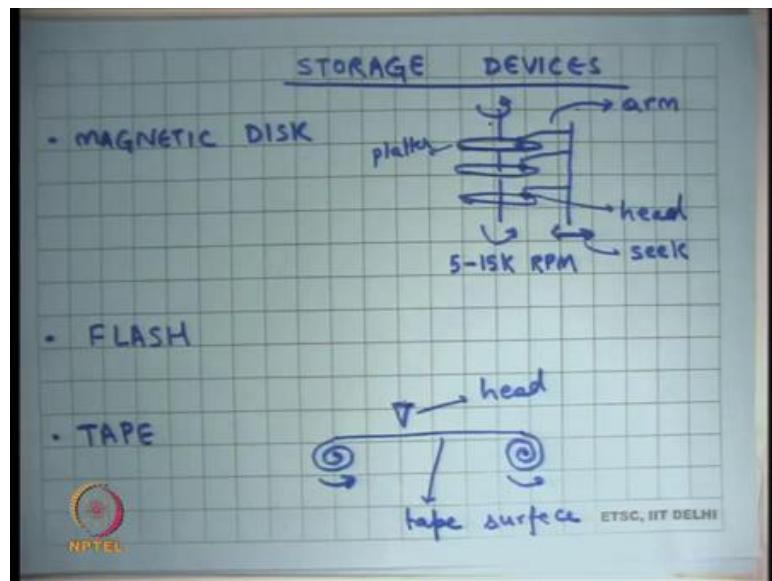
Let us stop here. And, start our discussion on file systems next lecture.

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

Lecture - 31
Storage Devices, Filesystem Interfaces

Today we are going to look in more detail at exactly how a file is implemented using the current set of storage devices, that we have and then what is the; what is a file system interface in general right.

(Refer Slide Time: 00:41)



So, let us look at the storage devices, you know predominantly there are three types of storage devices. In fact, you know the magnetic disk is by far the most popular storage device and we have already seen it. It basically has platters on a spindle, on a spindle and the spindle rotates at some speed typically 5 to 15,000 RPM.

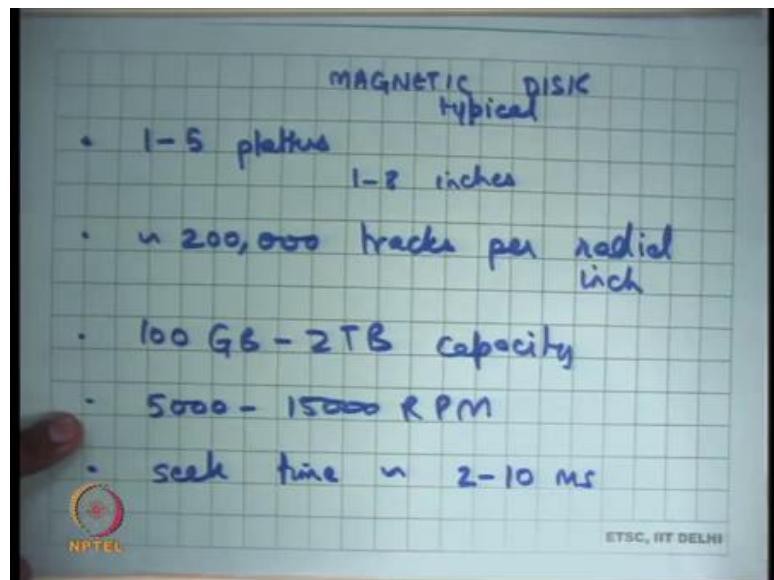
There is a there is an actuator arm which has multiple heads that are reading this data of the spindle of the platters. So, these are called platters, each surface is called a platter. And this these disk arms move readily alright while I mean it is possible to have separate discounts for different platters, you know in practice mostly you will have a single disk arm and multiple heads attached to the same disk arm.

So, all the heads move together in one direction or the other right. And, we also discuss some things about you know how long it takes to seek and in general what is the; what is the throughput of such a system, and we are going to discuss it in more detail.

There is another technology which is relatively more recent, it is called flash, which is available on phones, pen drives etcetera. It does not have any moving parts; it is basically made of solid state which means semiconductor except that its nonvolatile. So, unlike your DRAM which is your main memory this one is nonvolatile which means its state persist across power reboots right.

And, then there is this older storage device that was extremely popular in the earlier days called tape. And, this tape is you know if you seen a cassette player its similar, basically the tape moves. So, you wrap around the tape in the spindles and the spindles rotate in the same direction and those of the tape sort of moves like this and there is a head that reads off the tape ok. So, let us look at these the characteristics of these devices in more detail before we talk about file systems.

(Refer Slide Time: 02:51)

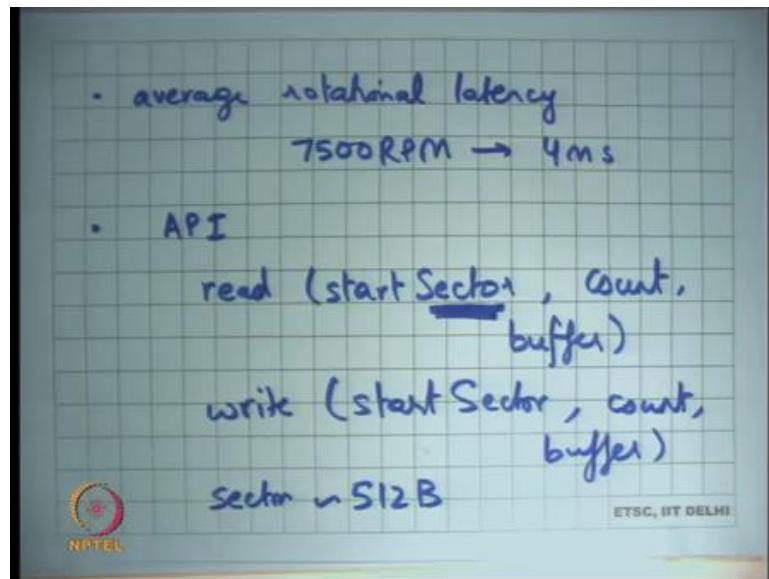


So, magnetic disk well typically 1 to 5 platters alright. So, typical magnetic disk let us say 1 to 5 platters roughly let us say 1 to 8 inches in height right. So, if you have ever seen a hard disk you will basically see that its a package which has you know which is roughly this thick 1 to 8 inches and you have 1 to 5 platters in it; roughly 200,000 tracks per radial inch right.

So, that is the density at which the tracks are packed radially, recall that a track is basically circle on the platter and so, they are concentric circles and each circle is called a track right. And so, the typical density today is 200,000 tracks per radial inch.

Today you can buy disk with 100 GB to 2 TB storage right, capacity right. We have already seen the rotational speeds, they are 5000 to 15000 RPM alright, seek time typical is 2 to 10 milliseconds alright and let us see average rotational latency.

(Refer Slide Time: 04:25)



So, the seek time basically means the time it takes for the head to reach the track that you want to reach and the rotational latency means the time it takes within the track to reach the sector that you need right.. So, on average the rotational latency will be the time it takes to make half a revolution right.

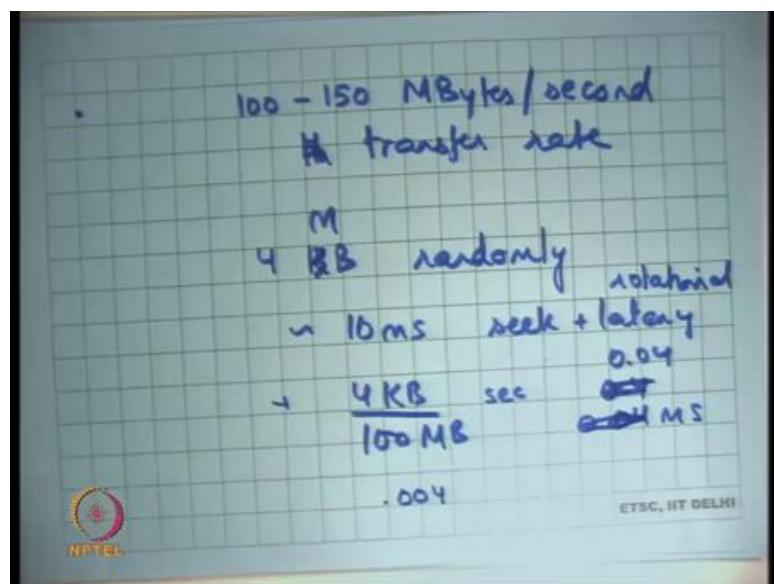
So, with let us say 7500 RPM disk implies roughly 4 milliseconds average rotational latency alright. And, what is the API of a disk or in other words you know how can you access the disk, how the how can the OS access the disk favor to write it in high level?

It basically you can say read, you can specify a start sector, account, and a buffer right. And similarly, you can do write same thing start sector, count, buffer; a sector is basically a unit of storage within a track and typically a sector today is 512 bytes. So, an operating system can ask the disk to read a sector or write a sector and the disk arm; so,

the disk controller will move the arm to the appropriate location and then read data of it alright ok.

Once the arm has been positioned at the sector from where you want to read after that it is just a matter. So, the disk the spindle is constantly rotating right. So, it is just a matter of the head starting to read the data that is stored in magnetic form on the platter. And so, in doing so, with the current speeds you will typically get 100 to 150 megabytes per second of bandwidth transfer rate right.

(Refer Slide Time: 06:33)



So, after you have positioned the head at the right position you know you can now start reading data at roughly 100 to 150 megabytes per second speeds alright. Of course, you know if you let us say so, what that means, is that let us say if I wanted to read 4 kilobytes randomly from you know somewhere I have to actually do seek and latency.

Then I will have roughly 10 milliseconds of seek plus latency where, latency, by latency I mean rotational latency right plus there will be some data transfer time which will be the time it takes for 4 kilobytes to go under the head. And, assuming 100 to 150 megabytes per second 4 kilobytes will probably take 4 KB upon 100 MB seconds right. So, that is.

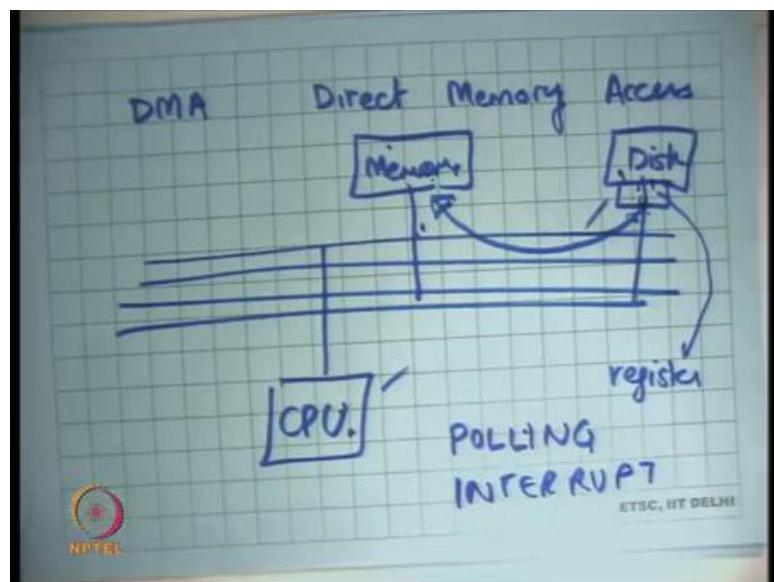
Student: 0.04 milliseconds.

0.04 milliseconds let us see; so, that is a 0.004 0.4 milliseconds right 0.04 milliseconds.

Student: 0.4 milliseconds.

All right its 0.04 milliseconds. So, as you can see its highly dominated by the seek and latency you know a random 4 kilobyte read is highly dominated by the seek and rotational latency. On the other hand, if you read let us say 4 MB of data then the you know then you are basically amortizing the seek and latency over a much larger chunk of data and this time becomes relatively more significant.

(Refer Slide Time: 08:33)



Finally, disks or other devices support what is called DMA or Direct Memory Access alright. What does this mean? Well, how does the CPU initiate transfer? So, direct memory access says that the device can directly access memory without needing the help of CPU to you know do this transfer.

So, let if I just draw the diagram again and let us say this is my bus and this is my CPU, and this is my memory, and this is my disk. Then one way for data transfer is that CPU one by one starts reading data from the disk. So, recall that the disk has some registers here right.

So, the controller will have some registers through which the CPU can program the disk. For example, by writing into the register the CPU can tell the disk I want to read the sector right and then it can read values from those registers to get the answers from the

disk. And, recall that these registers can be accessed either using port mapped space. So, port space like inbe and outbe instruction that we have seen before.

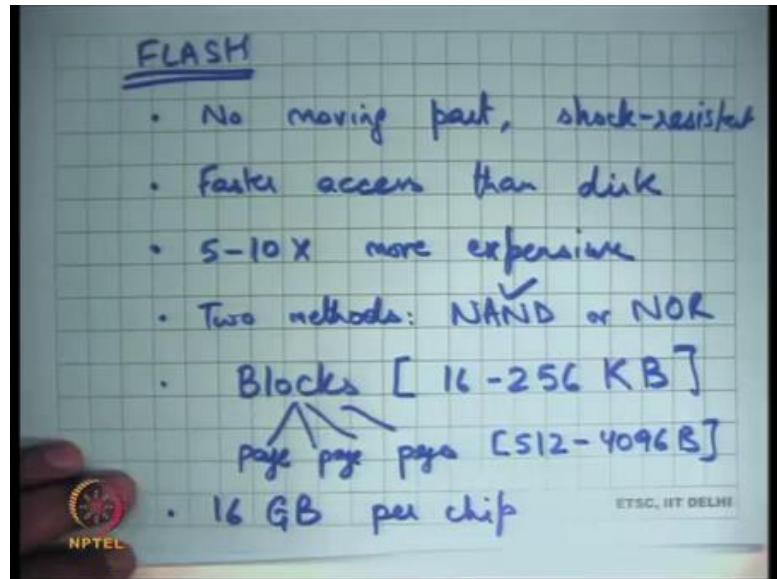
Or, they can be accessed using memory map diode right just load and store on some addresses that are mapped to those registers. So, in either case you are basically accessing the register. So, one way of doing transfer is that the CPU basically keeps reading these registers and then writing keeps writing them to memory.

So, it has a tight loop which basically read from the register and writes to memory, you know that would be a relatively inefficient way of doing it. Most modern disks have controllers that are capable of writing directly to memory.

The way it works is that the CPU registers a request and tells it that this is you know this is the location of the memory address. So, buffer basically points as a physical address that is given to the disk controller. So, and the disk directly writes to memory. So, the CPU just fires a command which says write these many sectors starting at this physical location in memory. And so, the disk controller in can do it without involving the CPU in it and so, that is you know you can imagine that a significant optimization.

And, typically that is how it is done; we have also seen that the CPU when the transfer is finished then the CPU can check whether the transfer is finished either using polling you know periodically keep checking whether the transfer is finished that is polling or interrupts alright. Or, you can configure the disk controller to generate an interrupt when the transfer is finished, and the interrupt handler will proceed accordingly alright ok.

(Refer Slide Time: 11:21)



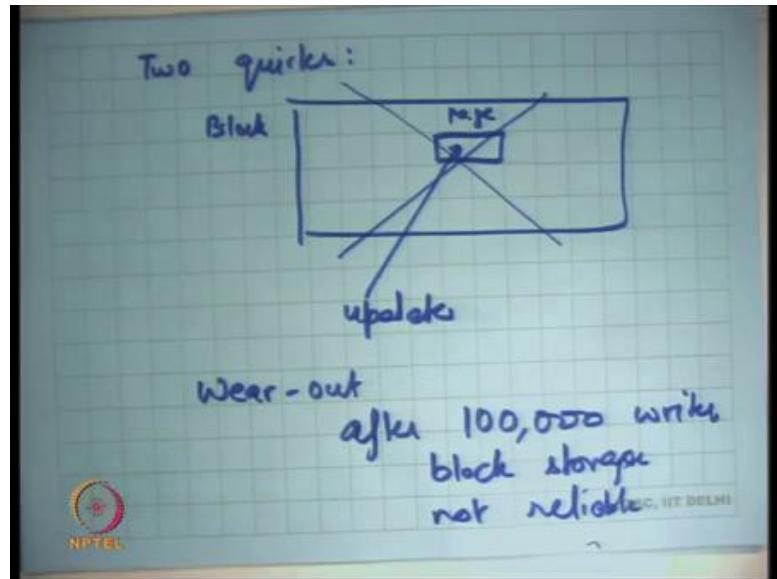
Alright. So, that is the disk, let us briefly look at the more modern technology which is flash and let us look at you know what it has. Firstly, it has no moving parts right. So, that is a significant difference, and which means it is you know shock resistant; you do not have to worry about it you know moving parts breaking down or anything of that sort.

Also, it has faster access than disk right, let us look at you know in what way that fast, but it is also 5 to 10x more expensive today right. So, two methods of implementing flash without going into details let us just say you know, let us just understand that there are two ways NAND or NOR and the NAND one is more popular today.

So, let us see how a flash device is organized, its organized into blocks and blocks contain multiple pages right. So, there are large blocks and the blocks are basically of the size of 16 to 256 kilobytes right and pages are 512 to 4096 bytes.

So, the page is basically is equivalent of a sector on a magnetic disk roughly speaking its between 512 to 4096 bytes. But multiple pages are within a block and a block is 16 to 256 kilobytes on today's flash. And, total capacity today that you can get for a flash device is around 16 gigabytes per chip alright.

(Refer Slide Time: 13:27)



So, there are two significant you know quirks about flash storage which were not there in you know magnetic disks. Firstly, if there is a block let us say this is a block and it has a page in the middle.

So, let us say this is page, this needs updation; updating a page within a block requires erasing the entire block before rewriting the entire block again right. So, just a matter of how its implemented at the physical level you basically need to erase the entire block and then rewrite the entire block to update some random location inside the block right.

So, that is that basically means that if you randomly want to write something on the flash a small write then it is expensive, and the second thing is were out. You can only write to a block let us say around after 100,000 times. So, after 100,000 writes block storage not reliable right. So, after you return to a block more than a 100,000 times you know the disk is no not reliable anymore. So, you know it has a limited self-life, these problems were not with the magnetic disk right.

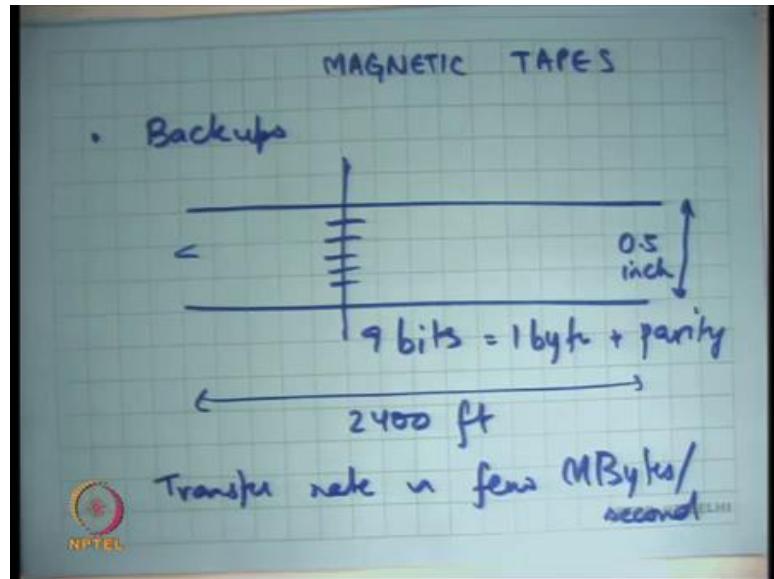
So, there is you know there is a huge amount of research and industry work going on to figure out if flash can be used as a replacement to magnetic disks for all the applications that. And, basically it involves solving these prob you know dealing with these problems in an efficient way.

So, for example, for wear out what you would want to do is you want to build some support either at the operating system level or at the hardware level to ensure that all blocks were out at roughly the same time.

In other words, there should not be any hotspots in your flash storage. So, you should not there should not be a blog that is constantly being written to right because then that. So, the disk is only the flash storage is only as good as last only as long as the block that first goes wrong right. And so, that is one optimization and the other optimization that people work on is basically that you know avoid random updates.

So, organize your file system, structures or your disk drivers in such a way that random updates are avoided ok. And so, you know there are so many ideas there that that can be discussed, but of course, that is not the; that is not the topic of our discussion today.

(Refer Slide Time: 06:15)



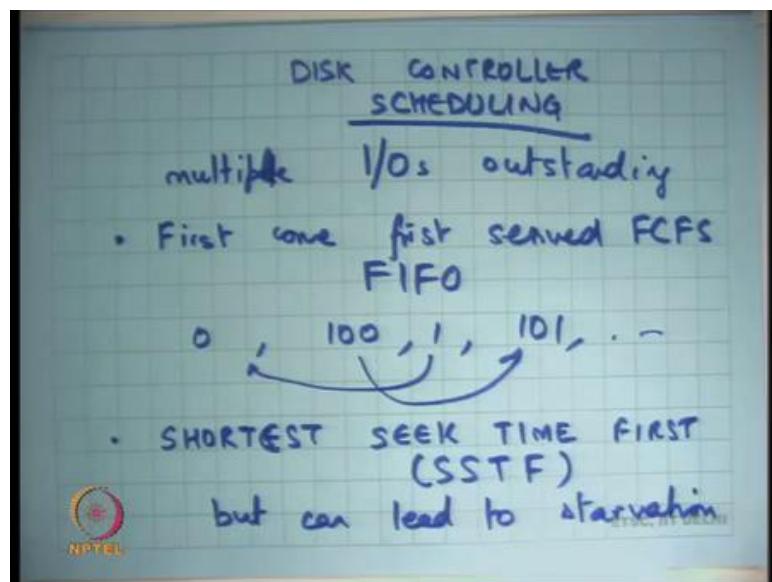
Let us also look at magnetic tapes right. So, magnetic tapes were used a lot for backups, till maybe early 90s or even late 90s ok, but not so much today. Today people use disks for backups instead of using tapes and the reason people use magnetic tapes earlier was basically tapes were much cheaper at that time than disks alright. So, let us see how a tape works. So, let us say this is a tape then it has you know 9 bits.

So, of storage in one cross section; so, its roughly 0.5-inch half a inch wide alright. And, the length can be up to 2 you know large feet 2400 feet let us say ok. And, each cross

section of a tape will have 9 bits which means 1 byte plus parity that is how it typically its used. So, for error detection you basically have a parity bit.

And typical bandwidth transfer rate is roughly few megabytes per second. So, as you can see that today's disks are far much faster than tapes and disks are become in those days disk used to be much more much more expensive. And so, backups which involve lots of storage were done on tapes and so, you would find you know dip lots of tapes in the backup room or in the server room. But today the same thing is instead done using disks alright.

(Refer Slide Time: 18:09)



Finally, let us look at how let us look at the disk controller in a little more detail.

Student: Sir.

And, in particular I am going to look at the scheduling algorithm that goes on within the disk controller. So, in case of a magnetic tape, if you want to access some random byte then you have to actually go through the entire tape to actually restart byte; assuming you have basically have variable sized records in your thing right.

So, that is the; that is the disadvantage; so, that is why it was primarily used for backups and not for you know online transaction processing workloads, alright. So, let us look at the disk controller. So, basically the disk controller has to decide assuming it has multiple IOs outstanding, it has to decide which IO to serve first right.

So, by IO I mean a read or a write request let us decide which IO to service first. And, it seems intuitive that the more flexibility the disk controller has which means that the more outstanding requests there are, the better job it can do what scheduling it right.

So, for example, one option is just First Come First Served right, or you know also called FCFS or also called FIFO First in First Out, right same thing. So, this is fair whoever comes first gets there, but what it completely ignores proximity. So, let us say there were alternate requests one for track 0 and another for track 100 and then there is a 1 then 101 and so on.

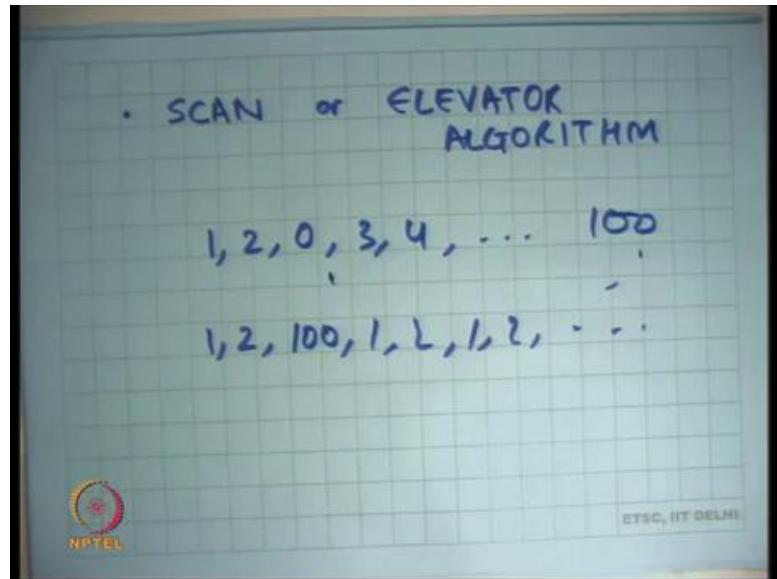
Then you are doing a lot of seeks right, you are spending a lot of time doing seeks; you the better thing would have been to do 0 and 1 together and 100 and 101 together right. So, that is basically a better thing to do FIFO is fair, but it completely ignores location. The other thing to do is Shortest Seek Time First right or also called SSTF that is it.

So, here the idea is let us say I have n requests in my queue, I will compute which of them is my closest to me is closest to the current location of the head and that is the one that I am going to serve. So, you can imagine that if there are lots of requests outstanding in the queue then you know you can get; you can get significant optimizations there.

So, this is good, but can lead to starvation right what can happen is that you lots of requests are coming for 0 1 0 1 and then there is an outstanding request for 100 which will never get served because the shortlist is basically always between 0 and 1. This is the part of the disk controller so, which means on the hard disk.

So, this scheduling algorithm is within the disk right. So, I am currently discussing you know how the disk controller is optimizing things within that package of the hard disk right. So, there is some logic within the hard disk that is going on to do all this ok.

(Refer Slide Time: 21:57)



And so, and the third one which is what is used is called the scan algorithm or elevator algorithm. And, it takes its name from exactly how typically an elevator works which is that you first choose a direction. So, the two directions right when you are moving readily either you will move inwards, or you will move outwards. So, you choose a direction in FIFO order.

So, whoever came first you choose that direction just like in an elevator, but if you have chosen a direction you will service all requests in that direction before changing the direction ok. So, it is very much like an elevator right. So, let us say somebody on the top floor presses a button and after that somebody in the middle floor presses a button then you know it will serve all everybody all the way to the top floor.

In fact, let us say there was somebody on top + 1 who presses the button with by the time it reaches top then it will also serve top + 1 before it changes direction right so, that is the idea. And so, that takes care of both efficiency and is also starvation free right. So, if you have chosen a direction after that you are going to go through and serve all those people in that direction. Of course, you are going to come you are going to change direction in some time and that is when you are going to. So, there is no request that can get starved in this case alright.

So, one way to think about the elevator algorithm is you choose the direction in FIFO order and within a direction you use shortest you use SSTF right. So, within a direction

you will choose whoever came comes on the way first right you want. So, with if you have chosen a direction whoever comes first you are going to give him. So, that is now within a direction is it SSTF and choosing a direction is FIFO.

Student: Sir.

Yes.

Student: Sir, would it be better to just keep continuing in one direction itself, since it is a best, we could eventually come down to 0 once again.

I am talking about the radial movement here.

Student: Ok.

Right, I am talking about the radial moment of course, the rotation just keeps moving, I am talking about the radial movement.

Student: Sir.

Yes.

Student: Even in this case starvation can happen, even in this case starvation can happen.

Student: So, these suppose a request comes from 1.

Ok.

Student: Then request comes for 2, then a request comes for 0 and then 3 4 5 6 7.

Let me write it. So, 1 2.

Student: 1 2 then 0.

1 2 0.

Student: 0 then 3 4.

3 4.

Student: So, on up to 100 maybe.

100. So, after 100 it is going to come back to 0 right.

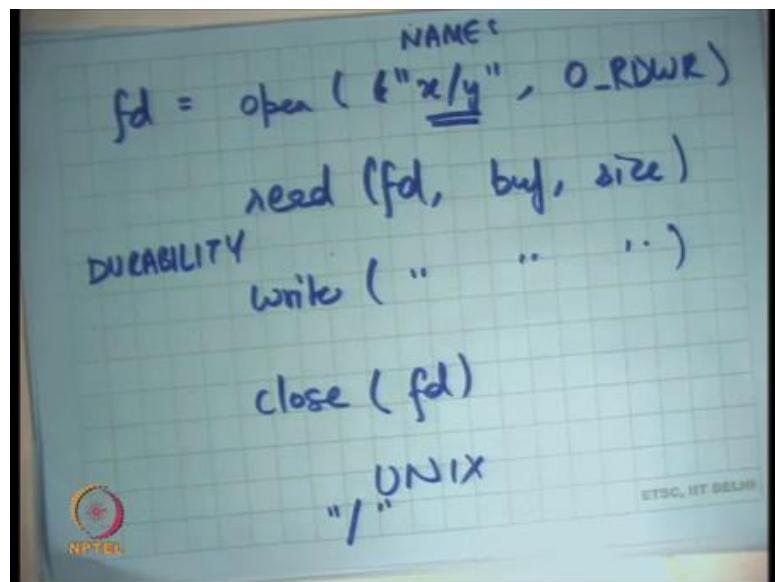
Student: 0.

I mean its it cannot go on for forever basically right; in the case of SSTF it could go on forever for example, I had 1 2 100 1 2 1 2 and so on right. So, 100 gets stuck forever basically, but the here there is a bounded time within which it is going to turn back; it by starvation I mean that it is never you know there is a possibility that there is a non-zero probability that it will never get served. There is an unbounded time, you cannot say when alright so good.

So, those are the storage devices, but by far you know in the last 20, 30 years the magnetic disk has been the primary way of doing storage, secondary storage. And, for that reason most of our operating system instructions have been designed and optimized for magnetic disk right. Tomorrow if you know with some inventions in physics you get some better storage, it also means that many layers above this in software and then hardware needs to be changed to better deal with these new devices.

So, let us now completely think about magnetic disk from now on and let us look at how what kind of abstractions that the operating system provides us.

(Refer Slide Time: 26:01)



So, recall that we have open some file name which is let us say a string x/y and some more which says you know read write right, that gives me a file descriptor. Then you

have read on file descriptor and some buffer and some size and you have write same thing and you can have close right.

So, open read write close is the interface that we are very much familiar with. This interface you know let us let us say this was you know Unix. So, this interface treats a file as a stream of bytes starting from 0 till you know some maximum length and that is it. So, there is no structure on the file so, it is completely unstructured.

On the other hand, you know the other option could have been that files are some have some structure. So, for example, you could have said my operating system provides files which are a sequence of records where each record is of size you know 64 bytes. And, the first field of that record is a natural number, just some 4-bit 4-byte number ok.

So, databases do this right; so, databases represent data as some structured data is to the some structure it is a collection of records and there is a structure on the records. As opposed to that an operating system has decided that we will not use any structure on the files, we will just read the files as a sequence of bytes that is all. If somebody wants to implement a structure, he can build implement a structure on top of this interface right. So, you can always say that this file you know you can always see the sequence of bytes, you can always interpret a structure on the sequence of bytes right.

The disadvantage of doing that, the only disadvantage of doing that is that you have two layers of reaction right. So, you are implementing a files as a in a certain way and then you have implementing a records on top of this virtual stream of bytes right. So, the file system has not so, the disk number of disk accesses and seek time, the notations have not been optimized for the record structure right.

On the other hand, your application may want to optimize it for the record structure and that is the reason that a database will not use a file system to implement its logic right. So, the database tries to bypass the file system the database wants direct access to the disk. And so, and it specifies what is this; what is this structured in variant that you want right. So, the only difference between so, that is the reason so, for performance reasons the database will not use a file system beneath it.

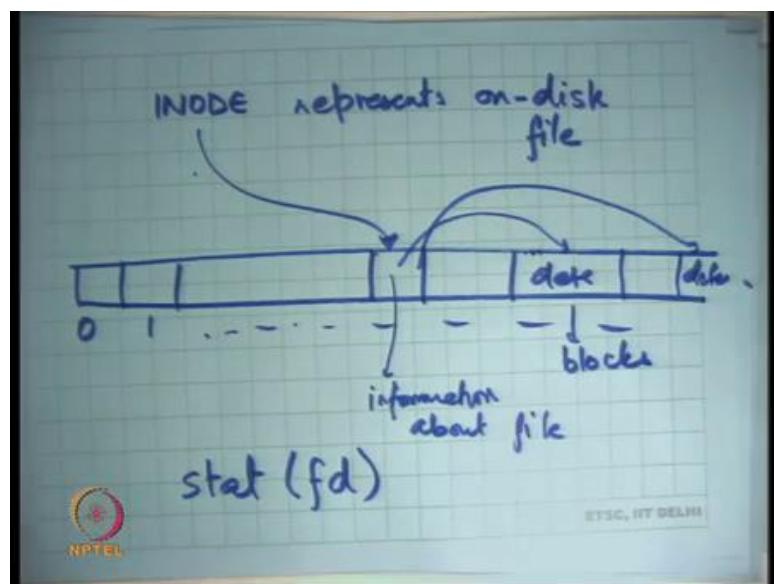
The database will directly access the disk and use structured representation right at the disk block level to get better efficiency. But, in either case you can always you know for

compatibility purposes run a database or a file system and in which case the file will be treated as just the disk basically alright.

So, let us look at this interface once more so firstly, there are names. So, the file system needs to implement naming right. So, on Unix you basically have the special name for it slash which basically means the root directory and then there is a concept of a directory which contains mappings from names to files alright and so, and you can do this in a hierarchical way.

So, the root directory so, there will be a special place on the disk where the root directory stored. And, the root directory will have a mapping from names to files and then those files themselves could be directories and so on right. So, it can be done in a recursive manner alright. What do I; what do I mean by names to file? So, what is the way what is the what is the file?

(Refer Slide Time: 29:49)



Well, a file on a disk typically is separate from its name; so, it is also called an inode represents on disk file, it is a handle to name a file ok. So, let us say this is my disk, I am representing it as logical namespace sector 0, sector 1 and so on right; till some value and a file is represented by an inode. So, inode will point to some location here which will contain information about the file, this is a one disk inode right.

So, Unix refers to a file by its inode and not by its path name, a path name translates the name into its inode and from there on its an inode that is important right. Inode is the on-disk block that contains information about that file. So, for example, when you do fd is equal to open some name and some permissions, it will obtain the mapping from this name to this inode.

And, then store a mapping internally between fd and the corresponding inode number right. And, then when you call read then it is just going to look at that inode and do the operation is based on that. So, there is something called an inode which basically represents represent the file and the user cannot see that inode except you know there is a system call called stat which allows you to. So, let us say stat on fd which allows you to inspect certain fields of the inode for example, what is the size of the inode and what is the status of the inode whether it is read only write read write etcetera and so on ok.

So, you can this structure is completely hidden from the user except that he can inspect it using stat, which is basically to just read write access, it cannot you know directly write to it. Writes to the inode by the user can only happen through this interface which is read write. So, for example, when you write something then you know something needs to be changed on the disk. So, the inode will get over written, but there is no other way to directly touch the inode by the user.

The file itself has you know the data of the file will be stored as disk blocks on the disks right. So, let us say this is data, this is data 2 and so on and the inode should say where this data is. So, it should have some mapping between offset in a file and the data on the disk right. So, there is data are called you know disk blocks and when you say read you basically implicitly there is an offset which basically says this is the offset value.

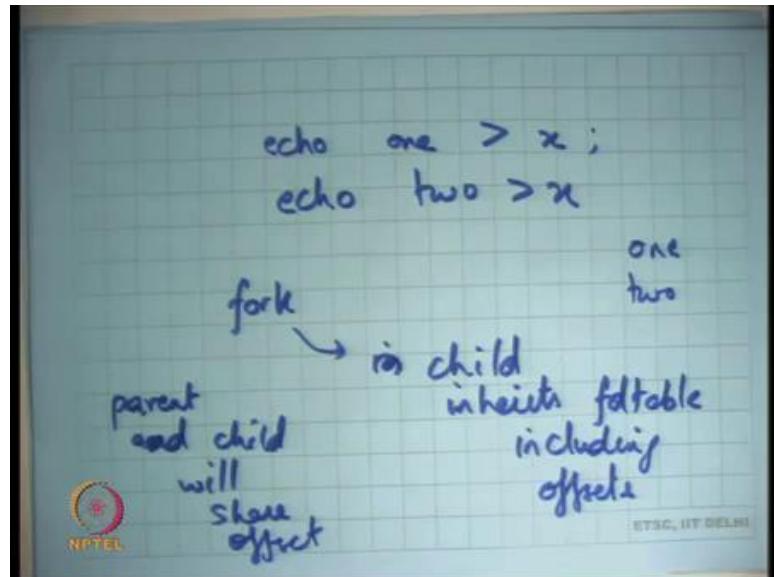
And, so the operating system should convert the offset into the corresponding disk block number using the information from the inode to get that data for you right; so, that is basically the idea. So, recall that in this interface when I say read fd, buf and size no where do I say a given offset right, the offset is implicit. When you open a file, you know depending on what mode you opened it, let us say if you open it in read write mode the offset is initialized to 0.

And, then as you read or write to the file the offset is automatically incremented by the data items that you read or write wrote right. This actually turns out to be an interesting

design decision is at the interface level. The other option could have been that you, you know having a fourth parameter here which says what offset you want to read.

And so, the user explicitly says this is the offset I want to maintain read and so, you know if he wants to read sequentially then it is his responsibility to maintain the offset in its local variable order ok. So, let us look at the tradeoff between having an implicit offset as it is done and Unix and having an explicit offset right. So, if you have an implicit offset then let us say you wanted to implement something like echo one to x and echo two to x right.

(Refer Slide Time: 33:51)



So, basically want to write to file to write one and then you want to write two right. If you had explicit offsets then they would end up overwriting each other right, because each program will not know. So, recall that when you fork you also inherit when you fork child inherits the fd table which includes the offsets right.

So, in this case the parent is going to fork a process which is going to execute the command echo, echo is an executable and its going to write something to the file. When it is going to write something to file because the offset was implicit and was shared between the parent and the child.

So, what will happen is that the parent and child will share the offset right. Because, the parent and the child will share the offset; if the child writes anything to the file the offset

of the parent automatically increases right. And so, you do not end up overwriting each other you end up appending to each other's output.

The output the appending append may have the non-deterministic, but in any case, it's you know it's not overwriting. If you can do some synchronization of this time that you know first you are going to write one, then you by semicolon you basically mean that you going to wait for the first process to exit, before you start on the second process.

Automatically will happen is the first the first child process incremented the offset and so, the parents offset also got implemented and then the second child process saw the increment saw the updated offset. So, you basically see one and two in the file right. So, it is possible to do this, if you had explicit offsets then you know to do this you will have a more complicated interface, will have to explicitly say, will have to you know pass another argument here that at what offset you want to write right. And, that offset needs to be incremented between two consecutive calls to echo right.

In general, so you may say, but you know it sounds it sounds dangerous if I spawn a child and its writing then my offset is getting incremented. And so, and I may not be expecting it, well I mean if you care about different offsets then one option is to just reopen the file right.

So, when you fork you basically copy the file descriptor table which means you have shared offsets, but if you want separate offsets all you need to do is reopen the file. So, implicit of you know if you want to explicit offsets it is easy to do it by just reopening, but the other way around is very hard.

So, its implicit offsets seem to be a better interface. So, it is a good; its a good example of you know how interface design can greatly impact your program structure and program elegance which I should say. So, this file API open read write close actually has turned out to be quite useful, it is not just used for accessing on disk files, it is also used for accessing things like you know pipes.

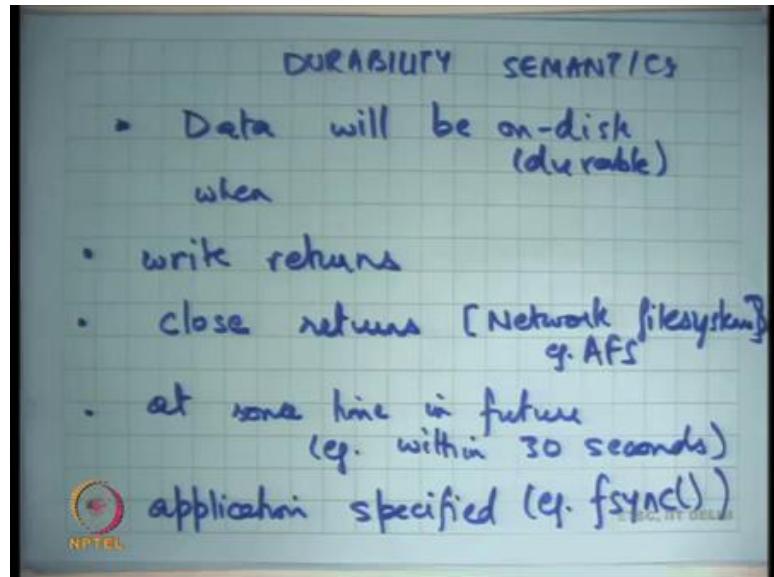
For example, you can call read and write on pipes, we have we have seen that before. It can be used to access devices; I can call read and write on the console device and it basically means the same thing alright or any other device for that matter.

And, also its it can be overloaded to do things like the slash block file system right, that you have seen before which allows you to look at the kernel state right. So, it's a very it's a useful abstraction that took some time to you know get concretized but has been used in Unix later plan 9 and then Linux.

And of course, you know other operating systems today. Now, let us look at; let us look at durability semantics of these calls. By durability I mean if I say right can I be sure that the data is on the disk which means if there was a power failure after I called right after the right you turn.

And, then you know there is a reboot can I be sure that whatever I wrote is going to be present on the disk right. So, these are you know these are the this is part of the semantics that the operating system provides to you and these are flexible. Unix does not specify what semantics you want, it is up to the operating system designer to decide what semantics is one. And, let us look at what are the different semantics that can have that you can have. So, let us say durability semantics right.

(Refer Slide Time: 38:39)



So, one is data. So, by durability I mean data will be on disk or you know durable when one option is when write returns right. So now, operating system may provide a guarantee that whenever the write returns you can be sure that the data is on the disk ok. This is great, its most conservative; basically, saying that I will always you know on

every write system call I am going to access a disk which means your buffer cache is basically write through right.

And so, if you are doing 1 byte writes then you are basically its very slow. So, it is very slow, it is most conservative. So, it is not; it is not; it does not look all that great, the other option is before close returns. So, you can say you know I do not care if there are multiple writes, but if you close the file that is the point I will say that you know definitely all the data for this particular file has been made durable right.

So, that is another option and some network file systems use this alright example: AFS. So, AFS is a network file system that uses. The other option is at some point in future; so, no guarantees exam. So, let us say example within 30 seconds right. So, the write may return, the close may return, but there is no guarantee that the data is actually on disk. The only guarantee is that within some bounded time, let us say 30 seconds your data will be on disk. But if there is a power failure within 30 seconds there is no guarantee right.

So, this has the highest performance, it is a right by cash; the operating systems cache replacement you know Daemon is going to run every let us say 30 seconds and it is going to flush things to disks. But there is no guarantee for the user that you know if you write something then its durable. But this does not seem like a very nice option right because after all you know imagine that you know your operating system was running inside an ATM machine and you wanted to withdraw some money.

And so, it deducts it deducted some money out of your account and it said write in your account. So, it's made the write system call and it return at its also made the close system call you got the money and you went away. And, what you did was you pulled out the power club power plug and so, you know that your update is lost basically.

And so, you plug it back again and you have the money back in your account. So, there has to be some way for the user or some way to for the application to say you know before I give this money, I should be 100 percent sure that the data is durable right basically.

And so, if you are doing this then you need some application level interface and this interface is called fsync on Unix. It basically says the application can call the fsync

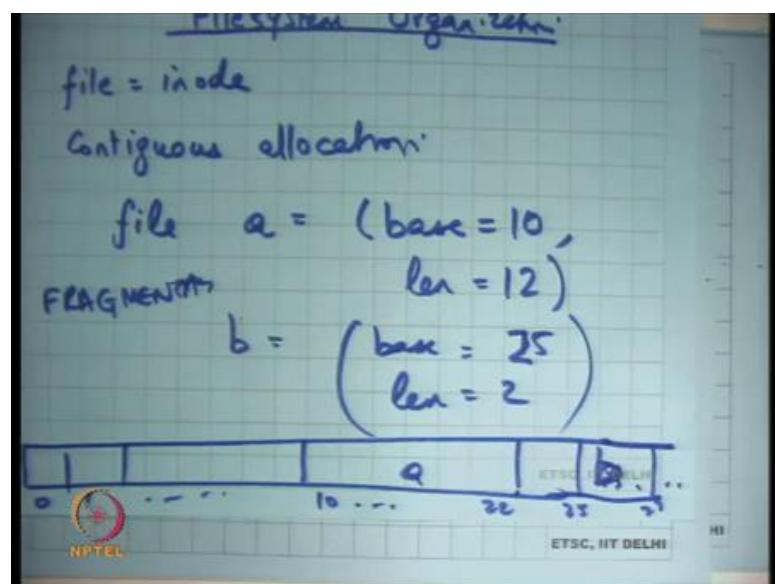
system call; so, this is again a system called on a file descriptor let us say or and then you say you know by the time fsync returns I will be 100 percent sure that the data has been made durable.

And so, for an ATM you will first call fsync, wait for fsync to return before you give out the money alright right. So, these are all sort of options with which the operating system with which the file system can work. And, something like Linux uses this right, at sometime in future with fsync right because of because that has the maximum ok. But, when you are doing something like this you have to be very careful because if there is a crash you should be in sure, that it does not corrupt the file system state itself right.

So, if you are doing some basically you are playing fast and dirty right. You basically saying I want to be as fast as possible, but that you know that should not mean that if there is a power failure at an inopportune moment; then you know your all your data has been erased or your file system has become corrupt.

So, it needs other ways to make sure that the file system remains intact and that is one of the most interesting design challenges of a file system. And, we are going to look at a few different ways of doing that and how modern file systems are doing it ok.

(Refer Slide Time: 43:35)



So, let us look at how a file is; so, file system organization right. So, before we talk about durability, we just got a sense of you know the here is this problem of durability and

semantics that the file system needs to solve. First let us understand how this how should the file system get organized right. So, by organized I mean I said that the file is represented by an inode, but then how should the data blocks be laid out. So, what should be the data structure that you use on the disk to basically store these files right?

So, one option is what is called contiguous allocation right. So, were file a is equal to some base, let us say base block is 10 and length which is you know. So, this here is a file which has which starts at base 10 and has 10 blocks right. And so, and similarly b is some base is equal to let us say 15 and length is equal to 2 whatever right. And so if I draw the disk then you basically have 10 22 is a and this cannot be 15 right.

So, it must be let us say 25 they cannot overlap right and then you have 25 27 which is b right. So, that is contiguous allocation right, this sounds similar to how we did segmentation in virtual memory right. So, basically you know there is a duel between files and processes and memory and disk. So, you basically say that entire disk is one big segment. This entire file is one big segment and you find space for. What is the problem?

Student: Fragmentation.

Fragmentation right.

So, for example, this space is cannot be used for a file which needs 5 blocks or something right. So, one problem is fragmentation. What is the other problem?

Student: File growth.

Growth right, if I want to grow the file a by 10 blocks, I have to copy the entire file to some other location and that is a global operation, that is a very very expensive operation. Because, to you know let us say by file of the gigabytes long and I want just wanted to append a few bytes to it and I cannot find space and I had to actually do you know a gigabyte operation to append a few bytes. So, that is not very good. What is some good things about these contiguous files allocation?

Student: Fast access.

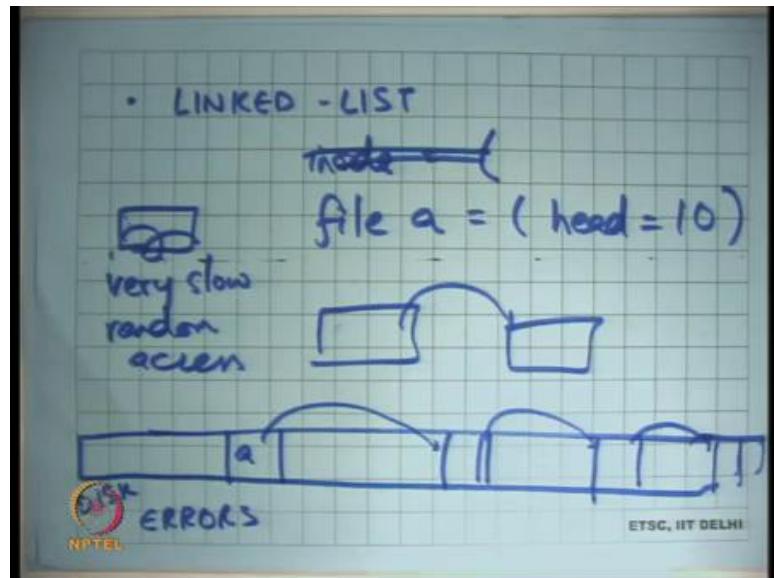
Very fast access if you want to know; so, the read system call, for example, only need to first get the inode. So, this information base and length it will be stored on the inode. So,

the read system call will get to need to get to the inode, get the base pointer, add the offset to it and then that is it then.

So, it is basically just 1 metadata access to get to any random location. More importantly sequential reads which is also a common case which means you know an application is just reading through the file sequentially, that is fast right.

A sequential file read directly translates to a sequential disk read and recalls that the sequential disk read is great right. So, sequential reads are fast here and so, this is great for any workload that requires sequential reads ok. So, we are not happy with fragmentation and growth. So, the other option is what is called linked list.

(Refer Slide Time: 47:05)



So, you know instead of an if it is an contiguous array let us have a linked list where inode; so, inode will point to let us say file a which basically means the inode storage is going to have a head pointer, head is equal to 10 and that is it right.

That is basically head is equal to 10 and then each block will basically say what is my next pointer right. So, if I have a disk then let us say this is file a its pointing to this location and this will have some next pointer which will point to b and so on right. And so, the storage that is been used for the next pointer will not be exposed to the user of course right. So, that is something hidden from the users.

The user cannot see the next pointer, it can only see the data which is stored in the rest 512 minus 4 bytes ok. So, what is what do you think about this? So firstly, access for a random offset is terribly slow, if I want to say I want to access offset 100 of let us say offset 10,000 of file a.

Then I have to actually do pointer chasing and at each pointer chase I have to actually get the disk in buffer cache and not just I have to get the disk in buffer cache and I have to wait for it to finish. I cannot overlap computation with disk access right, I have to wait for it to come before I can actually read its pointer.

So, basically, I have to block I have to keep blocking. So, random access is very slow, terribly slow; also, sequential access is also slow right. Because, now sequential access basically means that you have to actually do a lots of seeks because you have distributed all over that is fine. And, another important thing that; so, apart from efficiency the other thing you have to worry about in file systems is errors ok so, disk errors. So, for example, some blog goes bad, one blog goes bad in your file. What does it mean?

Right. So, in case of linked list if there is one blog that goes bad, the entire file gets corrupted right because you lose all the data. In the case of contiguous allocation one blog goes bad only that blogs worth of data has been lost, all the other data can be recovered right.

But, in the linked list allocation one block goes bad, the whole potentially the whole file goes bad right. So, you have to worry about that and also when you are doing this you know in ok. So, basically so that is another problem right.

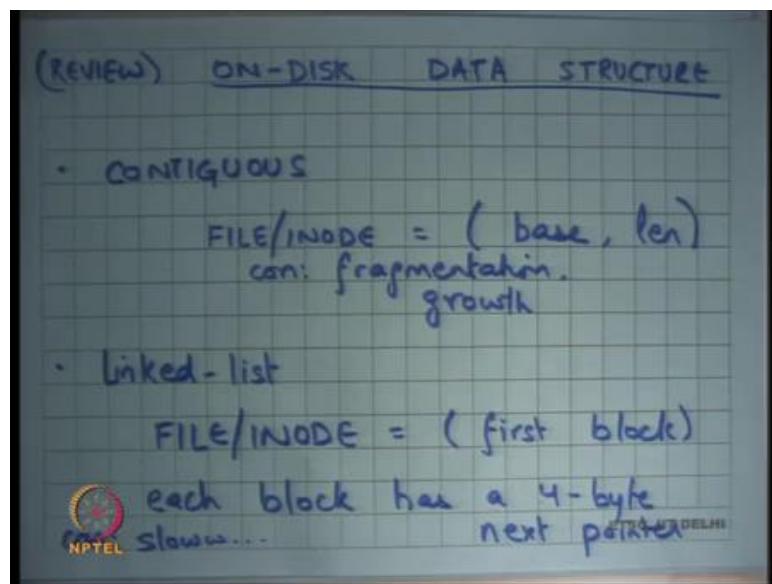
So, we have looked at contiguous file allocation and linked list file allocation and both seem rather impractical. Contiguous has fragmentation problems and growth problems, linked list has access efficiency problems. So, we are going to look at some more realistic file systems which are some combination of these next time.

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

Lecture - 32
Filesystem Implementation

Welcome to Operating Systems lecture 32 and we are talking about Filesystem Implementation.

(Refer Slide Time: 00:33)



And yesterday or in the last lecture we were looking at on disk data structures. We know which is basically the file system is nothing but in on this data structure. And we looked at 2 options one was a contiguous file system where contiguous allocation where a file or an inode is defined by or stores a base and a length and that is where the file resides. Then advantage of that is that contiguous bytes in a file are also contiguous on the disk, so it makes for fast sequential access of the file.

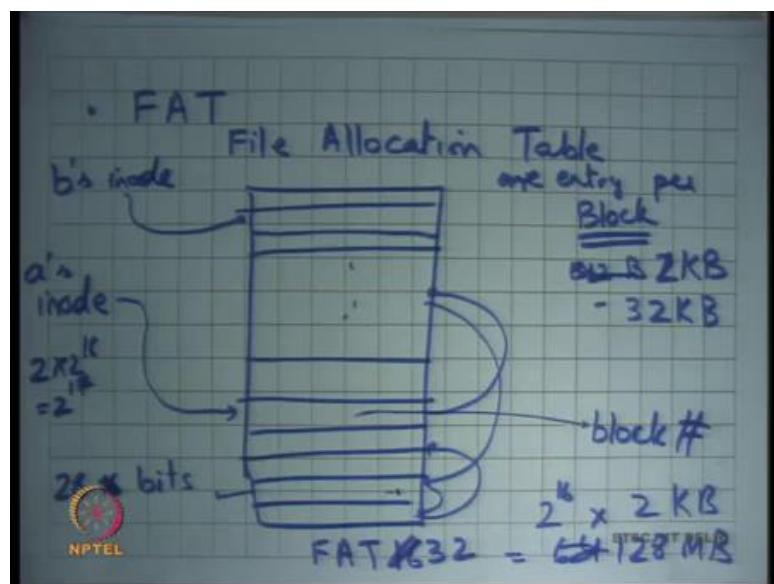
The bad thing about that here is fragmentation and growth right. So, it is similar to what we saw with segmentation in virtual memory it has those problems, the other on the data structure we looked at was a linked list which we said that let us the file or inode contain a pointer to the first block and then each block has a 4 byte next pointer. This gets rid of the problems of fragmentation and also growth, growth is very easy fragmentation is no problem, but it is very slow right.

It is slow not only because of sequential access, because you know for sequential access you have to just keep seeking because the file blocks will be strewn across all around the disk. But it is especially bad for random access you have to actually traverse entire linked list to get to any block in the file.

Moreover, notice that recall that our magnetic disk has this problem, that you know it is better for to give lots of IOS in flight simultaneously right. Recall that the disk controller is doing some scheduling internally and so it is it will be best for the OS to have lots of outstanding IOS for the days. So, that the disk and schedule them in the most optimal way.

But if you have a linked list allocation then and you are traversing the linked list you can have at most one IO in flight at any time right, because you have to wait for the first IO to finish and get the data before you get the next pointer and then you issue the next IO. So, there can only be one IO in flight and that is a really bad thing from a performance standpoint. And the third thing we said was bad about it was basically during reliability, so one link goes bad or one block goes bad the entire file gets corrupted.

(Refer Slide Time: 02:47)



So, let us talk about some more realistic file systems and one successful idea has been what is called FAT or File Allocation Table yes question.

Student: So, far the sigmatic segments if one block gets a corrupted end off if IO corrupted.

Student: So, going to that simple techniques like making that in case of storing both the head and tail can reduce a 12 bits.

So here is a suggestion that instead of using a singly linked list as I said let us have a doubly linked list and that would improve reliability by some degree. But really (Refer Time: 03:20) it I mean if you are just firstly it is it has all those problems of being slow right. So, that is no that is one big corner of this, the other thing is even if you have a doubly linked list if one block goes bad then the next point is gone, and the previous point is gone. So, I do not see how?

Student: I find it should be pointer of both the head and the tail, so in my inode.

Ok.

Student: I can reach.

I see, so you are basically saying that inside my inode I could just told the pointer to my inode. Let us say for each block I could store a pointer to the inode and so I know that these blocks belong to this inode.

Student: So, the what head on the doubly linked list and the (Refer Time: 03:55) of doubly linked list. So, this is a multi method 2 lock in the same file gets too low. So, means?

Student: He is saying like if a blocks gets corrupted; we can travel from page tails.

I see, so if the block gets corrupted then you can travel in the reverse order, means travel from the tail and you can organize your data structures are that 2 block. If only if 2 blocks get corrupted, then the file gets yes, it is a valid thing and you could do that but. So, it you know you could solve the reliability problem by adding more to this linked list idea.

But I mean it is bad anyways right it is bad because of performance reasons ok. So, here is an here is a very nice extension to the linked list idea, which is file allocation table first

used in MS dos, which was basically the idea is that this maintain this linked list not in the blocks themselves. But have a separate file that maintains this linked list right.

So, have a file which you will call the file allocation table right. So, this is the file allocation table or FAT, and this has all these entries for each block right. And now inode just points to so inode points to the head pointer which points somewhere here and within this file you have pointers like these right.

And so, if you want to go to so a file basically points to some entry here and that entry says that the first block it contains the block id right. So, block id or block number of the file and on the disk and so you can read that block so that is the first block. And if you want to get to the second block you change the pointer from here right. So, let us say if you wanted to get to the nth block you need to do pointer chasing within this table, you do not need to actually fetch the real blocks you only need to do pointer chasing within this table.

And the that idea is that if this if the table is small it can be brought into memory in one go and then this point is chasing can be done completely in memory right. And so, if you can do the pointer chasing in memory then you have eliminated much of the problems of the link less allocation that we had before ok. So, let us see how big does this file need to be well for every block we will have one entry.

Student: This block in a sector.

Right, so what does a block mean? A block would mean one sector or some small multiple of sector alright. So, maybe you know 8 sectors or 16 sectors, the MS dos block size ranged between you know 512 bytes to actually or 2 kilobytes for the FAT 32 system to 32 kilobytes right.

So, a block could be anywhere between 2 kilobytes and 32 kilobytes, the block size needs to be decided at the time of format right. So, a disk partition will have a constant block size throughout the life of that partition right. And so, depending on the block size you know you will basically name the disk blocks right.

And so, each entry here needs in so there the first incarnation of this idea was in the FAT 16 file system and the reason was called FAT 16 was each entry was 16 bits right. So,

each entry was 16 bits and the block could be between 2 kilobytes and 32 kilobytes. So, what is the maximum size of a disk that you can support, you know you could have at the most 2^{16} blocks in your disk and each block being let us say 2 kilobytes.

So, if you if the entry size is 16 bits the maximum size of the table can be only 2^{16} right that is the number of unique blocks that you can address and so that is 2^{16} blocks and into the size of the block. So, let us say it is 2 kilobytes then that is roughly 2 megabytes and 64 megabytes was 128 megabytes.

Student: 52.

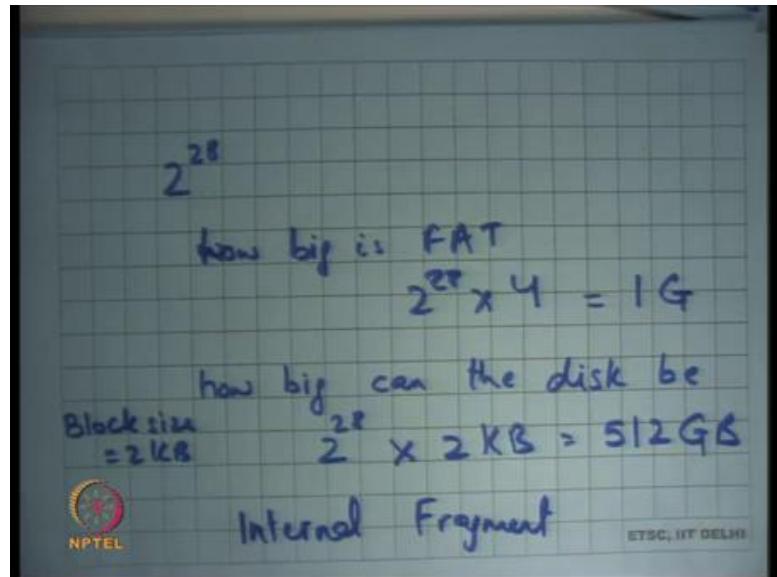
Right let us say the block size was 2 kilobytes and you have a 16-bit entry in your file allocation table you basically have a 128-megabyte disk that you can support with this organization. So, just to make sure that everybody understands this let us say I have another file. So, let us say this is a's inode and this is b's inode then b's inode with point somewhere else and then it will have its own linked list to follow right and one block will only be part of one linked list.

So, one block can only be part of one file there is no sharing between files of blocks right. So, for a one, so the FAT 16 file system with the 16-bit thing could support a disk of at most 128 MB does not seem large by today's standards. But at that time, it was a reasonable size given the technology of that time right, it is roughly late seventies early eighties.

And what is the size of my FAT 16 table 2^{16} entries each entry being 16 bits. So, that is what $2 * 2^{16}$ right that is 2^{32} or 2^{17} that is a 128 kilobytes right. So, 128 kilobytes at that time memories were typically bigger than 128 kilobytes and so it was a reasonable scheme to have alright.

So, but you know it does not make sense in today's technology, there was a so it was extended to what is called FAT 32 today. And FAT 32 had instead of 16 bits 28 bits for each entry and it supported block sizes between 2 kilobytes and 32 kilobytes and with this you can have a maximum file size of 512 gigabytes.

(Refer Slide Time: 10:20)



So, let us say you have 2^{28} entries, what is the size of how big is FAT $2^{28} * 4$. Let us say each entry is 4 bytes that is 1 gigabyte right and how big can the disk be? 2^{28} . Let us say I was using block size of 2 kilobytes, then I have $2^{28} * 2$ kilobytes that is equal to 512 gigabytes right. So, you can support a maximum size disk of 512 gigabytes with the fact 32 size file system with a block size of 2 kilobytes.

If you want to have a larger disk one straight forward thing to do is increase the block size right. So, you could take it up to till 32 kilobytes and that way you can know multiply this number by 16. Once again, the block size is decided at format time and just remains constant for the life of that disk partition.

So, what is the advantage of having a small box block size this is a large block size what is the tradeoffs what are the tradeoffs involved, one clear trade off is the larger the block size the larger disk you can support, but what are the other problems?

Student: It will.

Firstly, there is a you know what is called internal segmentation. So, I want I was interested in let us say 16 bytes you know in the of the disk, but I have to read the entire 32 kilobytes right. So, there is more extra things that you may need to read into the memory right, you need to maintain the buffer cache in that way and you so that is one

thing. So, you need more time for the data transfer also you have you polluter buffer cache.

So, the more data you bring in the more data you need to evict and if that extra data is not really useful then you are polluting your buffer cache. The flip side of it is the larger the block size the more spatial locality can be exploited right. So, it is really a tradeoff between spatial locality and internal fragmentation.

So, the larger the block size assuming your program has a lot of spatial locality, a better thing to do is to have a larger block size. On the other hand, if it does not have that much special locality you know the smaller block size would have performed better alright. What is the space overhead of a FAT file system?

(Refer Slide Time: 12:58)

The image shows handwritten notes on a grid paper. At the top, the words "SPACE OVERHEAD" are written in blue ink. Below this, there is a mathematical equation: $= \frac{\text{Amount of Metadata}}{\text{Amount of real data}}$. Further down, another equation is shown: $= \frac{4 \text{ bytes}}{2 \text{ KBytes}} = .2\%$. In the bottom left corner, there is a logo for NPTEL, and in the bottom right corner, the text "ETSC, IIT DELHI".

So, if I were to say you know space overhead of a file system, I basically say amount of metadata divided by amount of real data right. So, how much metadata is stored per unit of real data in this file system, well I am storing let us say 4 bytes per 2 kilobytes. Let us say I am using a 2-kilobyte block then that is roughly 0.2 percent right. So, that is not too bad that the space overhead of this file system.

(Refer Slide Time: 13:50)



What about reliability? So, let us first talk about Efficiency right. So, what were the few things that we were concerned about when we talked about efficiency number one, we wanted to say how good is sequential access. So, let us say I wanted to read the file sequentially from you know some x to $x + 100$ or whatever.

And so what should be my what should the speed be well in contiguous app it was great, what do you think in link less it was very bad what do you what do you think about FAT it is better than ling list.

Student: (Refer Time: 14:25).

Better than linked list because, you can compute in memory the list of blocks that need to be read a priori, you do not need to wait for blocks one by one right. So, you get you know the exact 100 blocks, these 100 blocks may be strewn across the disk. So, they may not be a contiguous necessarily, but you can give a hundred blocks on in flight to the disk.

So, you can have 100 outstanding IO simultaneously and the disk scheduler can schedule those 100 blocks efficiently and give you much better performance than the linked list allocation. But it is where it is not as good as the contiguous allocation of course, because the contiguous allocation would have take just you used one seek and one locate latency and the rest would have been the data transfer time.

Here you will have multiple seeks and multiple (Refer Time: 15:06) but at the same time you know because you have so many outstanding IOS the effect is amortized alright. The disk scheduler can schedule things properly. What about random IO? Let us say I just want to say get to the offset x in a file is it any is it fast or is it slow.

Student: Fast.

It is as fast as it can be right assuming that the file allocation table is in the memory you just chase the pointers inside memory, you figure out what this block you want to read and then you should that that requires just and now that is the minimum you need to do anyways yes right.

Student: Sir we need said about corruption of a block.

Right.

Student: (Refer Slide Time: 15:45).

Good.

(Refer Slide Time: 15:48)



So, that is efficient. So, we have talked about efficiency let us talk about. So, this is time efficiency of course and then let us talk about Reliability right. So, these are some parameters we are sort of evaluating our file system design on. So, is it reliable?

Student: Yes, provided memory is reliable.

The his has an answer provided memory is reliable, no we are not talking about reliability in terms of we are talking about reliability in terms of durability of data. So, you know it is we expect disks to live for multiple tens of years if not you know more. And so, if this cliff for that long the probability that one sector in the disk gets corrupted is extremely high, relatively high right.

And but we also want and the idea of using disk is that a data remains durable across tens of years let us say right. So, it is it possible that one disk block gets corrupted and that causes a lots of data loss in this organization.

Student: FAT.

Student: Sir.

Well if the FAT block gets corrupted. So, after the FAT is also getting stored on the disk right. So, that if the FAT block gets corrupted then yes you will basically suffer a lot of data loss. So, what is what is one easy way to fix it?

Student: Multiple.

Have multiple copies of the FAT that is alright. So, I have multiple copies of the FAT. I said 0.2 percent overhead now it will become 4 percent overhead space overhead.

But you know you will have you know significantly higher reliability right. So, that is easy so typically you will have multiple copies of 2 copies of the FAT in your on your disk to ensure the reliability question.

Student: Sir when we copy something or write something we will have to update on that table.

When we copy something or write something, we must update the table yes.

Student: But it can get the require right.

So, you know when I talk about reliability, I am talking about of a stationary disk all right. So, what happens if you know what happens while the program is running and some crash happens, you know that is not going I am not going to call it reliability I am

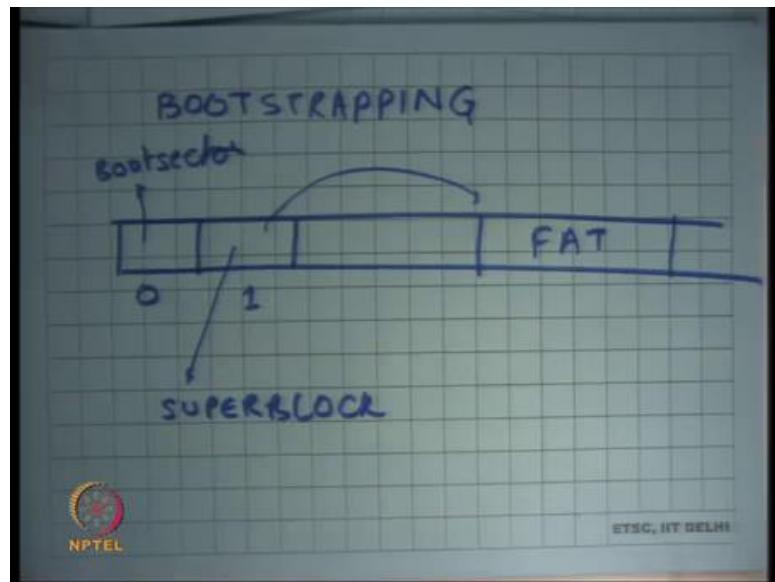
going to call it durability guarantees of the file system and we want to discuss that very soon ok.

But when I am talking about reliability I am talking about stationary disk and what is it is reliability. For FAT 32 we cannot keep the whole FAT table in memory, but yes you I mean you. So, of course, you don't need to keep the whole FAT table in memory, there is some caching that you can do there also. So, you can treat the physical memory as a cache for the FAT table itself alright and assuming there is locality of access in the FAT table the cache should have a very high hit rate.

Student: (Refer Time: 18:21).

Finally, how do you bootstrap the system?

(Refer Slide Time: 18:30)

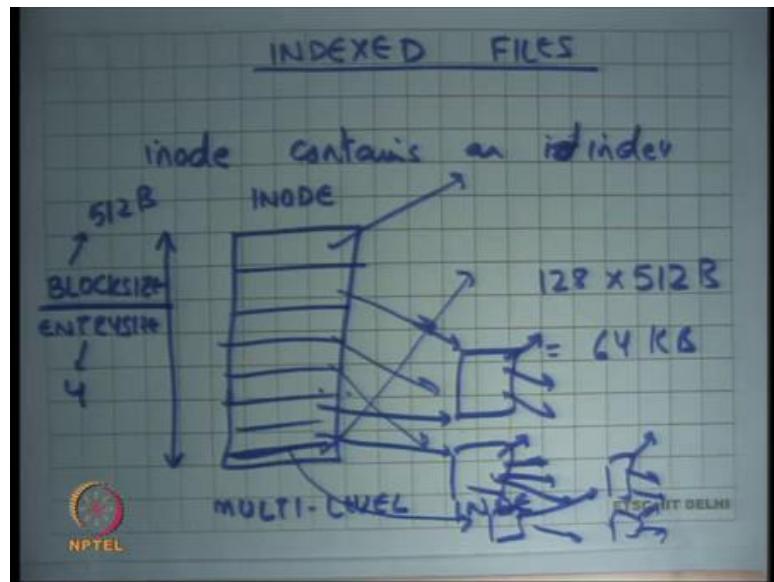


So, in general there for every file system. So, let us say bootstrapping so by bootstrapping I mean how does the OS know where the FAT lives right, once I get to the FAT then it knows where all the files live that is no problem. But how does the OS know where the FAT files well one you know one typical way to do that is let us say this is the disk and the 0th sector recall is the boot sector.

So, we do not use that for the file system at all ok. The boot sector is completely independent of the file system, it has nothing to do with the file system. Recall that the boot sector actually contains instructions that need to be executed at a boot time, so it has

nothing to do with a file system. And you know the first block is assumed to have some it is called a super block and this block contains information about the FAT let us. So, here is the FAT let us say right and then the FAT has pointers has to everything else alright ok.

(Refer Slide Time: 19:44)



So, FAT is a real file system that you use. The other type another type of file system is called what is called Indexed file system right. And the idea here is simple in the FAT file system we had the inode point to the head or the first block of the file. In an index file inode contains an index right.

So, here is an inode and it contains many rows and it has an index and so on right. Basically, saying that the first block of the file is at this location the second block is at this location and so on right. So, this is an index file. So, you have more information in the inode and so once you read the inode into memory then you can get to all the blocks of that memory of that inode ok. So, this is an alternate organization you just have an index over the thing. So, let us see what are the pros and cons of this.

Student: This index lives on disk.

This index lives on disk yes definitely.

Student: But these.

These are all on disk data structures.

Student: So, what then we are accessing the file give for the whole data structure into the memory and then.

Right, so now if I want to access a file and I know. So, this is called the inode right this index is also this index is stored in the inode. So, the inode needs to be brought into the memory and then you will look at the index to get the block right. Assuming that the inode was previously cached you know you get you take only one disk access. If the inode it was not previously cached, then you take 2 disk accesses one for the inode and one for the disk block right.

It is not very different from the FAT file system also you know you cannot you know it is not possible to store the entire FAT in memory or you may not want to store the entire FAT in memory all the time. So, sometimes you may take a miss on the fact itself right, so that it is similar to that ok. So, this is an index file system. So, what is the problem? Firstly, how big can the ion index b right.

So, how big can the index b you would want that the index is not bigger than the block right or the inode is not bigger than the block. And so, you know let us say your block size is block size let us say then block size divided by entry size and let us say entry size was. So, divided by entry size is the size of your index right. And let us say typical block size is let us say you know just take an example let us say 512 bytes and entry size is 4 then you have you know $512/4$ that is 128 entries.

And so, what does it mean? It basically means that there s a limit on the maximum size of a file that you can have right. So, what is the maximum size of a file you can have you have 128 entries into a block size of 512 bytes you know that is roughly $128 * 512$ that is 64 kilobytes. So, you know index files look good, but you know for example, for a block size of 512 bytes I can only have a file size of 64 kilobytes a block size of 2 kilobytes will increase the number by a factor of 4, but that is not good enough right.

So, it is not great because unless I you know increase the size of my inodes so, but so, it is a tradeoff between the size of the index and the size of the file. So, you know the larger in the index you can support, but that also adds to space over it. So, what is a

better thing to do, you know this is a one level index one level index can only grow to that you know one level index can only point to that much data.

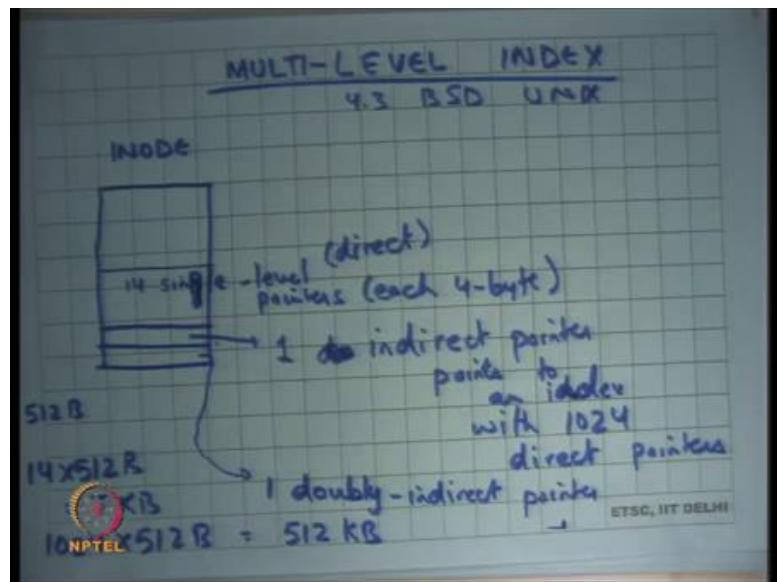
So, you want to have a 2-level index that is one way of you know we have all studied things like trees in our data structures goes. But if you have a 2-level index then the trade off is that I have to make 2 disk accesses before I actually you know actually get to the block, so actually 3 total disk accesses.

So, it is not and is given that most files are small right. So, most of the files in your file system if you think about it are executables which are roughly few kilobytes in size or configuration files which are just small text files or code files dot c or dot h files or java files these are also you know small relatively small files few kilobytes. And so, I want to optimize for the common case, which is lots of small files, but then I have some files that are really large right, like log files which run into megabytes hundreds of megabytes or even gigabytes right.

So, my typical workload or you know file size file distribution size this version is lots of small files some very large files and I want to optimize for that. So, instead of having a uniform 2 level structure what is done is you have a multi level index. What does this means is that the first few entries are single level, then few entries are actually double level which means they point to other indexes that point to more blocks and then the last entry or last few entries are actually triple level or actually this is called double level. So, then there is triple level which means you know it is a three-level hierarchy.

So, what you have done is that for the common case that you have small file, you have only one access in one meta data access and for the other common case which is very large file small number of large files you will have to make 2 or three meta data accesses. But those accesses are likely to be get degrade or mortised over a large number of blocks right after all there was a large file. So, you are going to probably access a lot of blocks in that file if you are accessing it right.

(Refer Slide Time: 25:46)



So, that is let me draw this again this is a multi level indexed file all right. So, let me take the example of a real operating system 4.3 BSD UNIX and let us look at x file system structure and users multi level index file inode contains. So, this is the inode it contains 14 single level pointers each 4 byte. Next entry contains 1 doubly index pointer or indirect block one in you know the other way to call it.

So, instead of calling it double index double level I will call it one indirect. So, these are direct pointers and one indirect pointer that points to an index with 1024 direct pointers all right. And last index so this is the this one entry and then last entry points to 1 doubly indirect pointer alright. So, by doubly indirect I mean it points to a block that contains pointers to more indirect.

So, it is a three-level hierarchy right. So, it is a 2-level hierarchy from here on, here there is a one level hierarchy and here these are direct pointers. So, with something like this you have optimized for the common case which is that small files. So, no small files of size less than let us say my block size was 512 bytes then files of size 14 into 512 bytes that is 7 kilobytes.

So, files of up to size 7 kilobytes will need only 1 metadata access to be referenced right. Files of size you know so 1 indirect pointers with 1 0 to fold direct pointer. So, that is 1024 into 512 bytes. So, that is 512 kilobytes. So, files of size up to 512 kilobytes will need 2 metadata accesses to be able to you know dereference them.

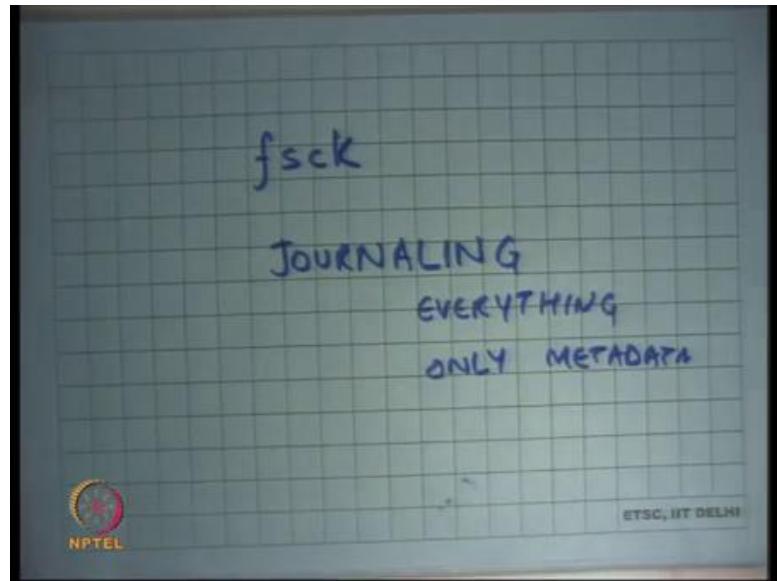
And then file is bigger than that which can you know which will be 512 megabytes or larger or can be done using 3 metadata accesses. So, that is basically it is done, still you have a size maximum size limit on the maximum size file you can have. So, you know every file system will have some limit and what is the maximum size of a file that you can have and, but all you are doing is just making that limit really large right and depending on the current technology that may suffice and or may not suffice depending on that right for example, you know for a long time Linux did not support files larger than 2 gigabytes right you know (Refer Time: 29:18).

For example, I remember till 2 early 2000s you know Linux was not supporting files greater than 2 kilobytes, but then now future versions of the file system basically allowed bigger files and so on right. So, depending on the hardware technology you will basically revise this as it goes.

So, that is basically the file system structure we have looked at the fact file system which is an in memory a linked list and then there is an index file system and both are widely used index file system is used in a Unix and also the NTFS file system is an index file system and the FAT that file system is also you know by its name called FAT file system.

Now, one of the things that you do worry about in a file system is what happens on a power failure or a crash you will you are in the middle of doing something and a crash happens and then you come back again your things may not be an inconsistent state, how do you deal with that all right. So, one way to deal with that is that you do not worry about it at all right, you just say if a crash happens in the middle of something.

(Refer Slide Time: 30:36)



Then I have add my at a reboot time I will run a program for example, you know in the Linux for Unix systems have this function program called fsck. So, you run the fsck program gets to run on a reboot and it will you know scan a global scan over your file system and see if it finds something wrong. And sometimes it may not be able to automatically decide whether what should the fix be.

So, if it finds something wrong what should the fix be it may not be able to automatically decide. So, it may need to actually ask the user and you know it is a very painful positive decision procedure for the user to figure out you know what happened and what could have what should the right thing to do.

And we are going to look at what kind of questions can come up and what kind of things that can go wrong alright. The problem with this the nice thing about this is that in the common case when there is no crash, things can work at full speed and very simply no not problem the bad thing about this is that the fsck program can take a long time. In the early days 90s disk used to be small so, a global scan was possible in a few seconds maybe you know 10s of seconds or minutes.

But will this sizes of 512 gigabytes to 2 terabytes you know distant and easily take up to know half an hour or 1 hour or maybe more ok. So, that is a cause of concern and so this technique while was used for a long time in real operating systems is no longer used and some other things are used the other. So, that is one thing, the other thing to do is

basically say that I will ensure that you know so there is something called Journaling and we are going to discuss what it means.

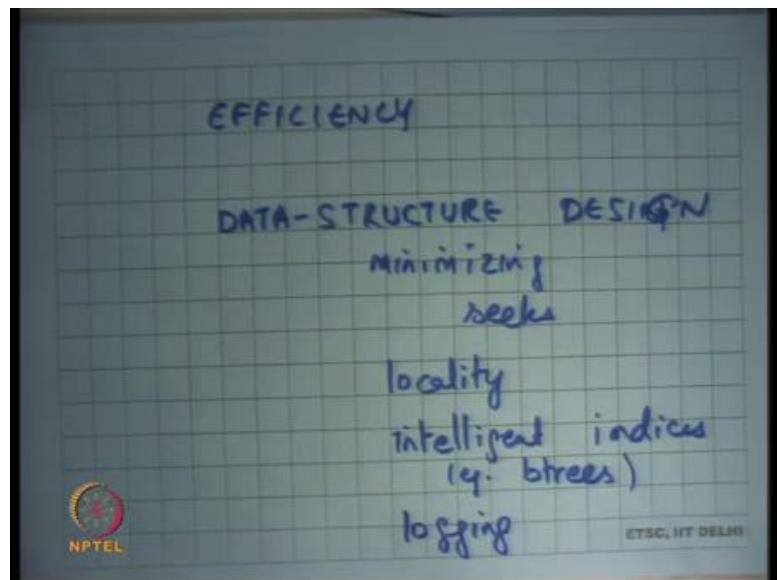
So, journaling basically ensures that your updates to the file system are atomic with respect to the power failure all right. So, there is a way to; there is a way to ensure that updates to the file system are atomic with respect to file power failure. So, if a power failure happened in the middle of an operation, then it is as though the operation never took place alright.

So, that is you know you maintain a data structure that may ensures that a non base data structure that ensures that the it does not take place or and if it (Refer Time: 32:40) happens after the end of the operation then you know the operation and so either the operation has taken place in full or not at all, so that is atomic. So, one thing is journaling everything, which means you general journal both the meta data which is you know these inode structures that you have and the data of the user that is that will provide you very good guarantees over atomicity and the user can be sure that whatever is doing is correct.

And the other option is to journal only meta data right. Here you are saying that I will make basically make sure that my metadata or the inode structures and other things that I am using for my file system are remain consistent across power reboots. But the users data may become an inconsistent right, it is something like you know killing a process. When you kill a process you make sure that the kernel side of state is cleanly freed right, recall that you do not just free the process you just wait for the process to reach some clean boundary before you actually free it.

But at the same time there is no guarantee for the user right the user could be in the middle of something and he can immediately get vanished right. So, it is a you and the user should expect that right. So, that is the other approach which is journal only the metadata and we are going to discuss this again.

(Refer Slide Time: 34:07)



The other thing you have to worry about is of course Efficiency. So, how do you provide efficiency? Well firstly, you know data structure design that we have already discussed for like them and this is basically about minimizing seeks right.

So, some things that can be done here is locality right. So, by locality I mean play try to place blocks consecutive blocks of the file consecutively on the disk, because assuming that the file has spatial locality it will automatically it will directly translate into spatial locality on disk and so you will reduce the number of disks this seeks right.

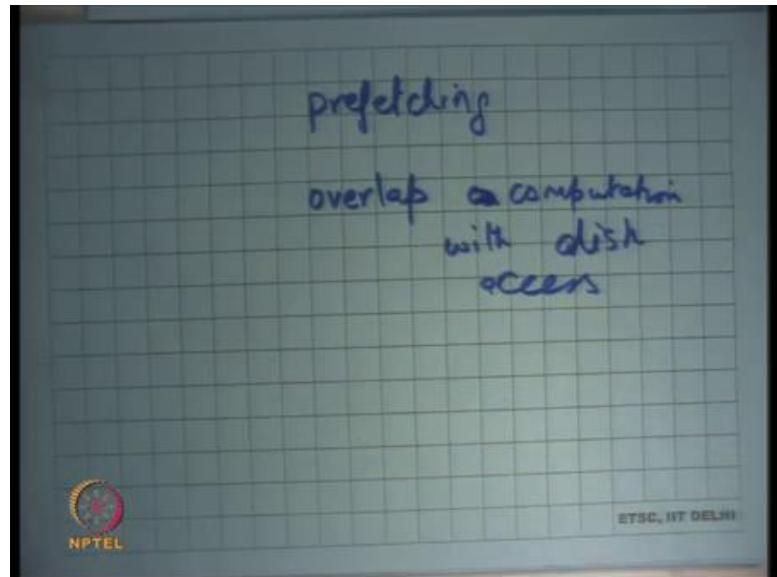
So, for example, your disk defragmenter program that you may have seen in some operating systems, will do we will try to improve locality to give better efficiency for a file system right. Then you know intelligent indexes intelligent indices right and examples you know multi level you have seen there could be you know you could use be trees and choose.

So, be trees with carefully chosen values of b can basically optimize performance because, your b can be tuned to the size of your block on the file system right. So, if you carefully choose b trees with the for the chosen b is then you can optimize these things and you know databases use this a lot right. And the other way to minimize seeks is what is called logging. So, here the idea is that try to make writes to the disk as sequential as possible right.

So, instead of right let us say I wanted to write ten different files at the same time and these files are completely strewn across the disk. So, instead of going to each file separately and writing to them could I you know organize my data structure such that all these rights appear as a log and so it becomes one sequential right and that is what is gets written to the disk and asynchronously it gets updated on the real file system ok.

And so, you and using that you can basically minimize the seeks required on the fast path which is you know when you are waiting for the district to finish. So, you can just log the data on to the disk in one go and asynchronously transfer it to the file and then we go to look at examples on how to do this.

(Refer Slide Time: 36:46)

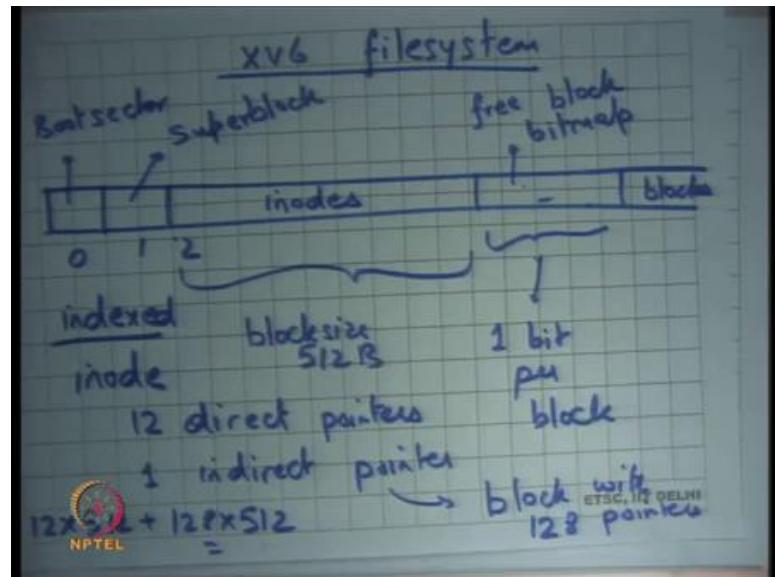


The other thing you can do is Perfecting of course right. So, you read some data you have some algorithm which basically figures out that there is a lot of spatial locality. So, you prefetch data of course there is a; there is a; there is a trade off trade off about how much to prefetch and how much you cannot prefetch and overlap computation with disk access.

It is a good idea to you know while you are doing the disk access to also keep computing, so that firstly you will keep you of CPU busy and secondly your computation may lead to more disk accesses and that may lead to better this scheduling overall all right.

So, we are going to look at you know. So, these are the basic sort of design principles on which we are going to decide what is a good file system and it is not a bad file system.

(Refer Slide Time: 37:47)



But let us first start with a simple file system which is the XV6 file system and let us look at a realistic implementation of a quite simple way of bone file system all right ok. So, so here is the XV6 file system, let us say this is my disk and this is block 0 that is the boot sector this is block 1 that is the super block right and then from block 2 to some number they have inodes ok.

Then to some point they have what is called a free block bitmap and then you have the block stem cells right, so let us look at it. So, the first 2 sectors are clear good sector and super block, the next few sectors are reserved for inodes. It basically means that all inodes will be allocated from here right. So, all the so that basically immediately puts restriction on the maximum number of files that you can have in a system all right.

One advantage of putting all the inodes together like this is that often you may want lots of locality between your inodes right. So, let us say you are doing ls right. So, ls basically wants to look at all the files in a directory and all it cares about is you know things like last modified time and some characteristics that can be just go you know you do not need to look at the data blocks you only need to look at some statistics about that inode right. And so, if lots of inodes are being read there is a lot of locality between the inodes and so you can service a lot of requires from here all right.

But on the other hand conversely you also expect there to be a lot of locality between inodes and data blocks right it is quite also quite likely that when you access an inode then you are going to access the data block corresponding to that inode. But this organization does not exploit that locality right so there is a trade off. Then there is a free block bitmap it basically says which all these blocks are free.

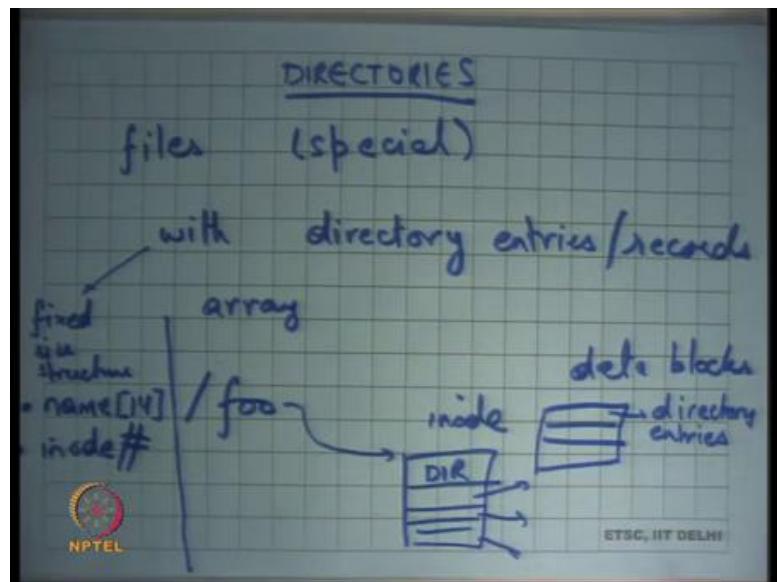
So, it is basically maintaining a list of free blocks right and basically using one bit per block, indicating whether that block is free or not. Once again you expect that this free block bitmap is a very sort of hot region because, if you are creating or deleting a lot of files or and so on or writing to a lot of files then you probably need to do that have this and so it is good to have this in a very local way. So, there is a locality in free block bit right.

Let us look at the inode structure. So, they use index files right and the inode structure has 12 direct pointers all right and 1 indirect pointer block size is 512 bytes. Indirect pointer points to a block with $512/4$ that is 128 pointers ok. Notice that the inode itself has only 12 pointers $12 + 1$ 13 pointers.

Where the indirect block and have more pointers why? Well the inode has also extra information in store like you know things like what is what the permissions are, who owns it what is the last modified time stamp and other things right.

So, there is more information in the inode. So, that takes up some space, so you have only space left for 13 pointers in the inode, but in the indirect point block you have the whole space available for pointers all right. So, you can calculate what is the maximum size of a file that you can have, you can have $12 * 512$ bytes + $128 * 512$ right, that is the maximum size of a file you can have roughly 64 kilobytes all right ok.

(Refer Slide Time: 42:33)



So, what about Directories, how our directories implemented? Directories are nothing but files but special, they are special in that the user cannot read them directly it is the operating system that can read it all right and so and it is only the operating system that can interpret it right. So, directory is a file with records. So, directory entries right and on XV6 it is represented as an array of directory entries all right.

So, for example, if you have a directory let us say you have a directory called slash foo, then slash foo will point to an inode. There will be some status file status flag in the inode which says I am a directory right, all other files will have this status flag saying I am a file, but a directory. So, that so there will be a bit which say distinguish begins files and directories.

And then just like everything else it will have pointers direct pointers. So, these are, and these are data blocks, but these data blocks will contain directory entries right. So, basically of a directory is nothing but a file which stores directory entries in them all right and directory entries and XV6 are represented by a structure which is fixed size structure which contains 2 fields name and inode number all right.

So, a directory entry contains 2 fields the name of the file or the directory right. So, this name could be the name of file or a nested directory and the corresponding inode number and XV6 says you know I will only allow names up to 14 characters long. So, you have a fixed size on how big the name can be and inode number is fixed and so there is a fixed

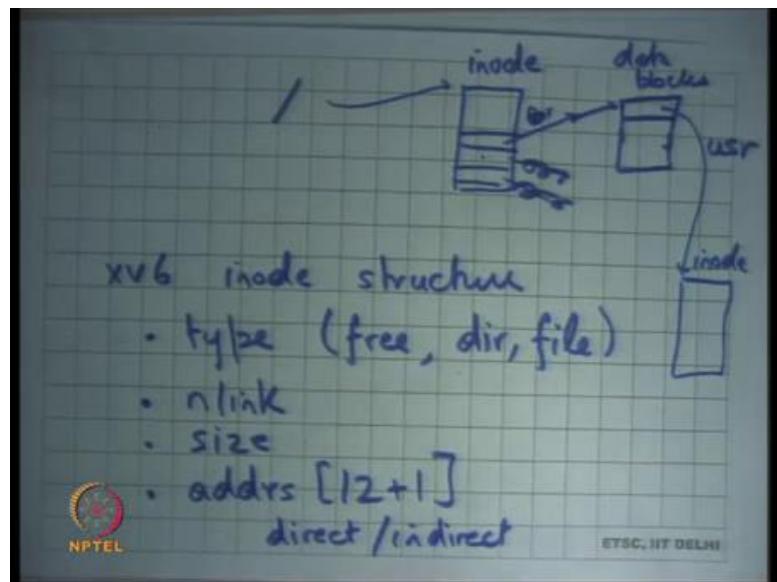
size structure of directory entries and that is basically you know used to implement directories. So, what is the maximum size of what is the maximum number of files you can have in a directory.

(Refer Slide Time: 45:30)

$$\text{max \# of files in a directory} = \frac{\text{max size of file}}{\text{sizeof(dirent)}}$$

Let us say my directory, so the maximum number of files I can have in a directory is basically is what? Max size of file which we computed earlier which was let us say 64 kilobytes on XV6 divided by size of directory entry right let me call it dirent. So, that is basically the maximum number of files you can have in a directory right. Whatever the maximum size of file you can support that is the amount of space you can have for storing directory contents and so you can have that many entries inside a directory.

(Refer Slide Time: 46:23)



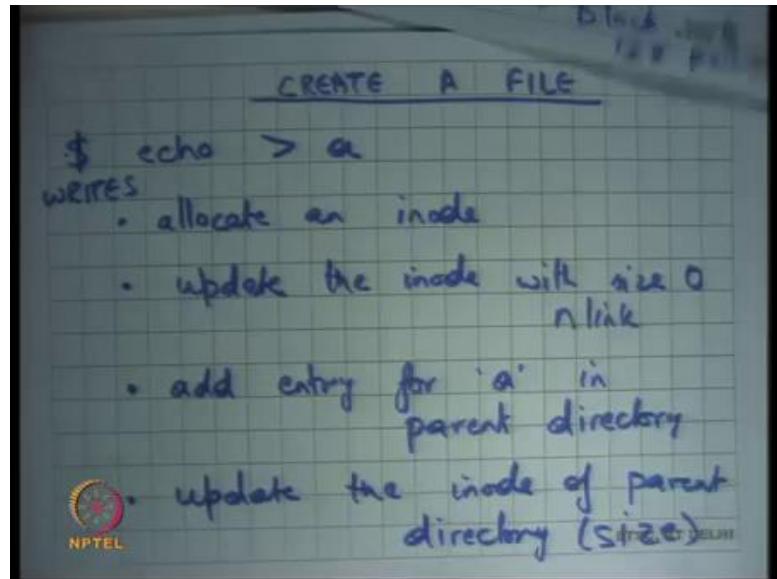
So, if I were to draw a diagram let us say this is slash. So, that is the slash directory the root directory that points to an inode right and these point to different. So, these have you know the name can be let us say use slash user it is pointing to another inode sorry. So, these point of data blocks right. So, the inode themselves point to the other box. So, let us say the slash directory has only 10 files and 10 files can fit in 1 data block.

So, you only have 1 data block all the others are 0 right and this data block will have directory entries which will be mapping names like slash user to inode numbers right. And they themselves may be directories which mean they themselves may now have this structure within them or they may files, in which case these data blocks do not have any special meaning they are something that the user has defined.

So, let us look at the XV6 inode structure once in detail a little bit, you have a type which says whether it is free or directory or file right. You have nlink which says how many inward links do I have right, what is the size? The size basically means what the size of this file for a file. And for the directory it basically means the number of directory entries into the size of the directory entry right.

So, in both cases size basically means the size of the data blocks or the size of the useful content in this file or directory and then you have the addresses or pointers you recall the 12 plus 1 thing right direct and indirect.

(Refer Slide Time: 48:47)



So, let us say I wanted to create a file. So, let us say you want to create a file on XV6 and let us say you did that using this command called echo greater than a. So, what happens now just as a recap you are running the shell process you type this command it calls folk it calls exec echo, echo program gets to run with its output redirected to the file a right.

Now before, but before the but a let us say does not exist earlier. So, the first thing the program needs to do the shell program use to do is to create the file a right. So, let us look at from a disk perspective what all needs to happen for the file to get created ok.

So, let us look at this figure I want to create a file a. So, first I will allocate an inode how will I allocate an inode? I will go through these lists of I nodes and find the first one that is free, so that is allocating an inode. So, that is one disk access and once when I allocate it, I will also mark it as a file instead of a free flag I will mark it as a file flag.

Then I will update the inode with size 0 right. So, you will allocate an inode and you will update the inode with size is equal to 0 and you may want to update other things like nlink etc. For example, now there is a directory that is pointing to me. So, you would not want to increment the end link of this file ok.

And you may also so you may also want to 0 out all the direct and indirect pointers. So, you know so that is another disk access you could have potentially combined the first 2 disk accesses if you are; if you are; if you are being smart about it, then you add entry for

a in parent directory right. So, how do you do this, if you know you basically look up the parents directory to a directory is file and so the file contents are going to live somewhere here.

And there you basically append to the directories file and another directory entry which has name equal to a and inode number equal to whatever you allocated, what do you found earlier right. So, that is another disk write. So, I am actually just counting the number of disk writes, I m not counting the disk reads at all right. You may actually need to go through the directory that is inode and the directories data blocks to basically get to that point and then you update the directories data block.

So, in this case the updation of the directories data block means adding or appending to it is contents. So, what is the last thing you need to do update the size of the directory right this you have appended to the file of the directory. So, you need to update the inode of parent directory particular size.

So, at this point you have created the file, you have allocated an inode initialized it created in mapping using by appending a directory entry to the parent directory and updated the size of the parent inode ok. So, you know many disk accesses for one operation all right. So, right so let us stop here and we are going to continue this a this method of looking at what happens on a file operation and in particular we are going to look at what happens if people can if multiple processes come concurrently try to write to the same file.

For example, or how does data get appended to a file and how do files get removed from a directory. So, you know what are all the operations there need to be done to do this ok.

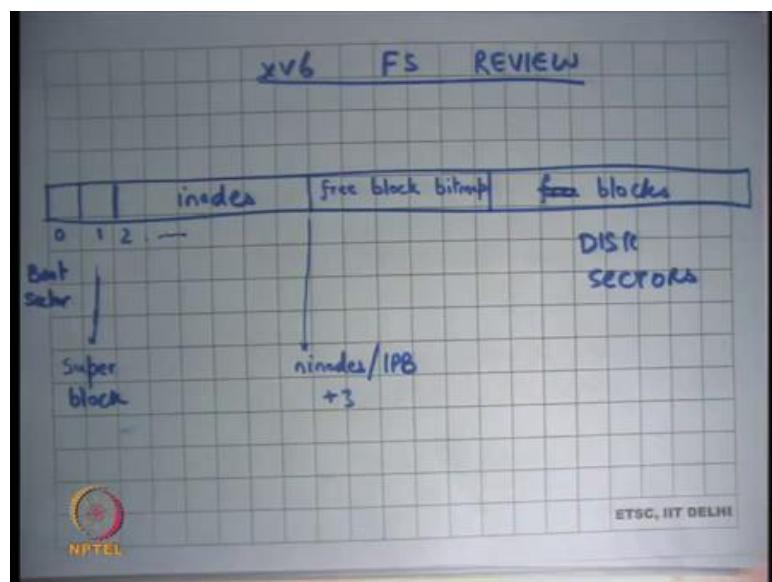
Thanks.

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

Lecture – 33
Filesystem Operations

Welcome to Operating Systems, lecture 33.

(Refer Slide Time: 00:29)



So, last time we were looking at Filesystems and how file system could be organized; we looked at few different types of file system organizations and we said let us look at the xv6 file system in more detail. Recall that the xv6 file system is an indexed file system in which each file is represented by an inode and an inode contains pointers to the data blocks right.

And, we said that you know this has problems of the maximum this limits the size of the maximum file. So, one simple way to deal with it is to have a 2-level hierarchy but given that most files are small having a 2-level hierarchy for all files is not a good idea.

So, the common optimization is that the first few data pointers or the first few block pointers are direct pointers. So, it is a one level hierarchy for the first few bytes of the file and after certain size of the file you have a 2-level hierarchy and still further you have a 3-level hierarchy. So, that allows you a very large file overall yet having a very

small access latency for small files which is the common case. Even for large files the average access latency is small because the cost of index lookup gets amortized over a large amount of data hopefully, alright.

So, we were looking at the xv6 file system and the xv6 file system the let us say this were the disk sectors right or disk blocks. The zeroth sector is the boot sector that is basically completely independent of the file system; file system does not touch it.

The sector number 1 is the super block that contains meta level information about what this file system is. For example, for a fat file system it may contain things like what is the block size right. So, this is sort of written at the format time.

And, then from sector 2 till some value which is hardcoded you have what is inodes right. So, the idea is that you only going to allocate inodes from this area. So, that immediately puts a limit on the maximum number of files you can have in the file system right and then there is the region which is the free block bitmap you can you basically store 1 bit per block to indicate whether the block is used or free.

And, then you actually have the blocks themselves; actually, I should not say free blocks these are basically you know all the blocks free or used blocks right. So, that these this is the place where all the data is stored; this is the place where you basically stored whether the block is valid or not free and this is the place where you store inodes for the files right.

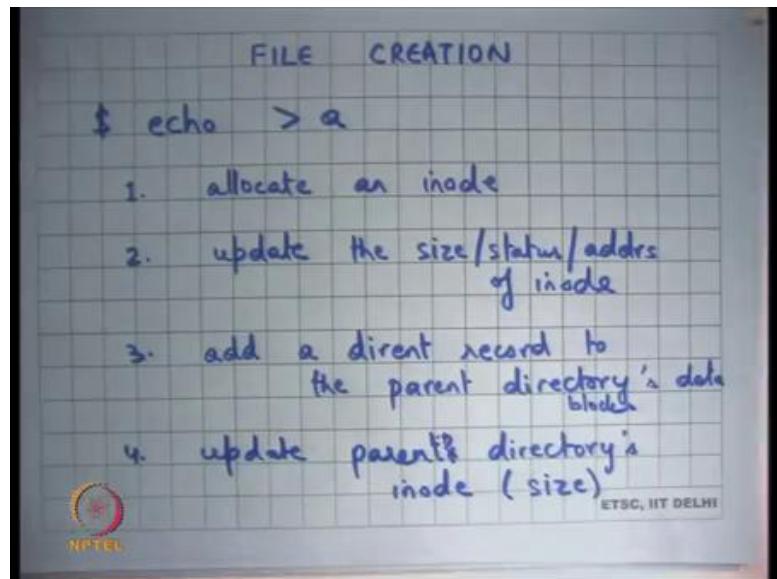
There may multiple organizations you could do here. You could have you know for example, interspersed inodes and blocks that would have had the advantage that you could have had probably better spatial locality between the files index and the files data blocks.

On the other hand, if you want to go through a lot of files for example, you want to do you know a scan of the entire data directory structure and you all you care about is looking at the inode of each file. For example, you just want to count the number of files let us say, then this is a much better structure because all the inodes are located in this region.

So, you can do one sequential read assuming your memory is large enough to read all the inodes and then process them together right. So, given that we are implementing such file systems on a magnetic disk the most important optimization in general is to reduce the number of random disk accesses, be it reads or writes right because a random access involves a seek kind of rotation. A sequential access is relatively much cheaper right alright.

So, and this constant IPB basically says what is the how many inodes can you store per block. So, inodes per block that depends on size of inode you choose. If I recall correctly the size of an inode on xv6 was 64 bytes and the size for block is 512 bytes right.

(Refer Slide Time: 04:33)



So, we are also looking at what happens if I want to create a file. So, let us look at some operation then look at what are the rights in the disks that need to happen for each operations. So, if I want to create a file let us say I create a file using this command echo greater than a, that just creates an empty file called a in the current working directly or the process right.

So, that current working directory is also the parent directory of this file called a. So, how I am going to do it? I am going to first allocate an inode, I am going to traverse the list of inodes and find one which is not used ah. Update the size, status, and adders of the inode, alright. So, basically just allocate the file and you know say that it is basically size 0, its status is allocated, read – write who owns it for example when it was created and so

on whatever else you want to write about it and then you create a directory entry record in the parent directory right.

So, you want to you have created a new inode, but you also want to link it to the parent directory and this case the parent directory is the current working directory. Linking it to the parent directory basically means adding a directory entry record in the parent directories file. Recall that a directory is nothing, but a file with the with the contents of the file being directory entries that indicate the files that are present in the directory right each directory entry is a mapping from a name to an inode number right.

And, you know for simplicity xv6 allows only a 14-character name at most. So, that ensures that a directory entry is the fixed sized directory entry. But, of course, you could make it more complex to allow better you know longer names for example.

So, you create an you add a directory entry record I am going to call it dirent record to the parent directories data blocks right. So, this block is going to be added to the directory entries or directories data blocks and then you going to update the parent directories inode.

So, you have appended a block to the parent directories file and so, you want to also update the parent directories inode to indicate that the size is changed right. So, you know 4 or 5 or some number of writes random writes to the disk to be able to create a file like this right.

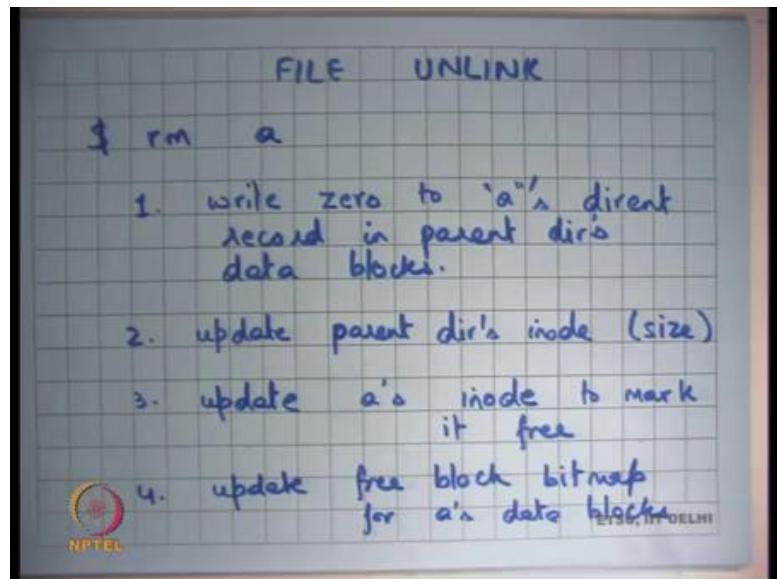
Let us see what happens on a file append. Well, in some sense file creation was similar to saying I want to append to a directory right. Creation of a file is nothing, but saying append to a directory, but let us see you know what happens on a file append. Here I instead of allocating an inode I allocate a block first which basically means write to the bitmap.

Blocks are implemented differently from inodes there you have separate bit map and you have the block themselves as opposed to the inode was having a status filed inside that whether it was free or not right. Then, you zero out the block contents you write X to the block. So, all this is basically being written in the block area.

This write was to the bit map area and these two writes are out to the block area and then finally, there is the write to the inode area that is basically you update the a's inode right. You update the a's inode basically you why do you need to update it? You need to update the size there is one character in it. So, you update the size from 0 to 1 and then you may you also want to update the pointers.

Recall that is an index file system, you are going to update the point inside the inode to point to the first data block right. So, once again all these are random writes they are completely in different parts of the disk and you are going to have to do 4 or 5 random writes to be able to implement file append in a file system like this alright.

(Refer Slide Time: 08:07)



Let us look at file unlink. I want to remove a file from the current working directory, or which is which should be of the parent directory of a and let us assume that a exists in this current working directory. So, now one way to do it is you first write you first look at the current working directory and find a file called a right.

So, you find something if you do not find something then you can return an error or if there exists a file called a. So, you find the file called a and you zero it out which basically means that it does not exist anymore right.

So, basically xv6 is implementing directories as a list of as the file that contains a list of dirent records, but the dirent records could themselves have a you know have a zero to

indicate that it does not exist. So, write zero to as direct record in parent directories data blocks. So, this is the write to the data block right.

Then, you update the parent directories inode. You may know in this case if it was a last one, you may want to update the size of the parent directories inode. So, first you wrote to the data block region, then you rewrote to the inode region and then you update a's inode to mark it free right.

So, this you are deleting a you have marked unlinked it from the parent, but you also want to mark it free, so that it is usable by other files or other operations in future and then you update the free bit bitmap block for a's data blocks, alright. So, you also mark the data blocks to be free, so that once again a few different writes that are happening to the disk right ok so, alright.

So, we get some sense of you know how costly these operations are to do read and write you know create read and write etcetera. Let us first talk about what do you deal with concurrency. So, let us say if there are two threads or two processes who simultaneously execute want to append to a file or they simultaneously want to create two files with different names in the same directory.

So, one needs to have concurrency management. One simple concurrency management is have a global lock over the entire file system. Does this global lock need so, you know you can have one global lock that is called the file system lock and only one thread can take this lock at any time by the definition of lock and then each the thread that completes its operation then gets out, but that is obviously, not a very nice design.

Anyways the file system operation is also slow and if only one thread can be inside the file system then if there are multiple threads accessing completely independent parts of the file system you have greatly serialized and made the execution and made execution really slow right. Recall that the most important one of the most important as optimization is to allow the disk controller to have lots of IO's in flight.

If you have one single serializing block for the entire file system you can only have a few may be even just one IO and flight at any time. So, having one global lock is not a great idea.

But, let us first talk about if I had a global lock where will I store this global lock? Is it enough to store it in memory or do I need to store this global lock on disk? Do I need to implement locks in memory, or do I need to implement locks on disk? What does it mean to implement locks on disk?

It looks implementing locks on disk means you know same thing you assume that something on the disk is atomic. So, recall that we said that there is some instruction that can atomically set locations in memory, bytes in memory right.

Similarly, we need some assumption on the disk. On the disk let us assume that right of a sector on the disk is atomic. So, if you know either the sector gets written or it does not get written. It is not like half of the sector gets written by one thread and then the half gets written by another thread, that is not going to happen. Also, let us assume that if the power goes down in the middle of write into the sector, then the sector write will get completed right.

So, the last sector will either get written or not get written at all, right, that is atomic. So, back to my question do I need the file system lock to be implemented as a disk variable as an on-disk variable or does it have to just have an in-memory variable to implement mutual exclusion? All I need to do implement is mutual exclusion right. So, can I do implement can I implement mutual exclusion by in memory variable or on disk variable? Answer.

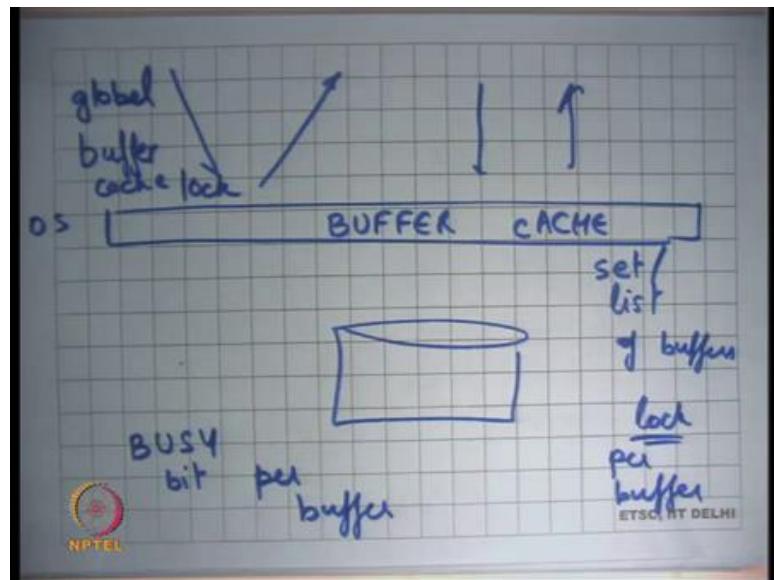
Student: If we want synchronization to proceed even after a power reboot.

If I want synchronization to proceed even after power reboot, well, I mean. So, right now I am not I am just saying that I want mutual exclusion across multiple threads and threads just die after power reboot right. So, threads are basically just in memory constructs. So, they do not; they do not carry over across power reboots.

So, in memory suffices if the only you know so, if you are basically assuming that your disk is only accessed by this operating system that you are using it is not this disk is not shared between multiple machines you know in memory locks are enough right. Assuming that all accesses to the disk are going through this OS layer, the OS is using in-memory locks, two synchronize accesses through the disk. So, in memory locks are enough all right.

So, that is fine. Now, let us look at, but now we are not happy with a global lock. So, we need per file logs, per inode logs, per block per data block logs or something of this of that sort alright ok. So, how can this be done? Well, you will want to have one log per every data block on the disk somehow alright.

(Refer Slide Time: 14:01)



So, one way to think about it is that. Let us say this is my OS and this is my disk. All accesses to the disk go through the OS and the OS inside it maintains what is called a buffer cache you seen this before right. So, a thread cannot read or write to the disk directly. The thread can make a request for a block to get read into the buffer cache and then you synchronize over the buffer cache right.

So, you can only read or write blocks that are present in the in the buffer cache. So, if you want to read something you will first bring it into the buffer cache and then read data from it, if you want to write something you will first bring data from first bring it into the buffer cache then write to it and you may want to implement write through or write back cache that is a different matter.

But, in any case every you know you have if you have started the disk blocks as a buffer cache and, now you need to synchronize on the buffer cache instead of synchronizing on the data disk blocks themselves because you can nobody can actually touch the data blocks directly. You can only touch the data block by bringing the data into a buffer inside the buffer cache and then you synchronize over the buffer cache alright.

So, basically you have a buffer cache which is you know some set of buffers let us say it is implemented as a list and each buffer of buffers and each buffer has a lock per buffer. So, if I want to let us say create a file into a directory. I need to make four operations.

What I will probably want to do is I will first lock let us say let us look at this let us look at this. I need to do four operations. These are four different sectors, four different blocks. So, I will probably want to lock all of them and then do this operation. Recall fine.

So, basically, we are moving from coarse grained locking to fine grained locking there is an operation that I want to perform. This operation needs to be atomic with respect to all the other threads that may be doing this operation. So, recall that the way to do it is basically if fine grained locking, identify all the data all the data items or datums that need to be touched and get all those locks a priori and then do your operation right.

So, that is what I am going to do get these locks. So, get an get a lock for the buff for the sector or for the buffer that stores the inode, get a lock for the buffer that stores the data block of the directory that contains the directory entry and so on and then release those locks. So, that is fine grained locking. How do I implement these locks? Should these be spin locks?

Student: No.

Do you think it is a good idea to have them as spin locks?

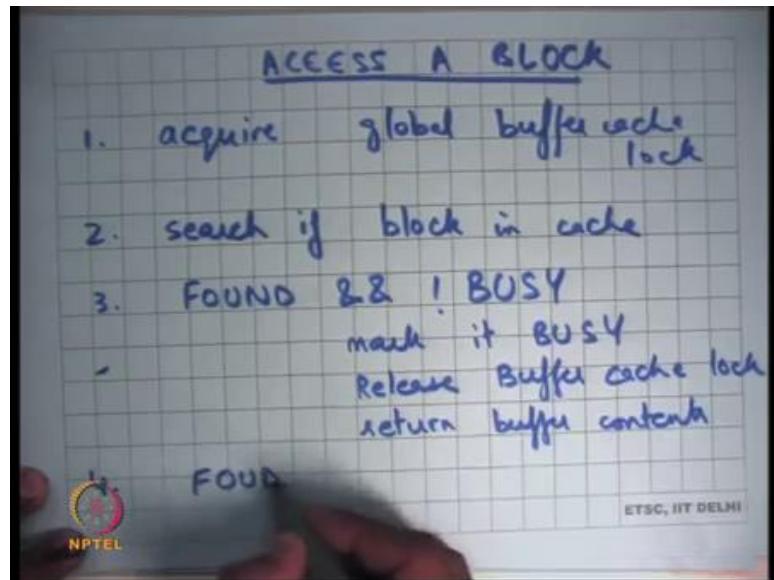
Student: Blocking lock.

They should probably be blocking locks right I mean if you have spin locks then if I am waiting for some buffer to get free or some buffer to get released, then I have to just I want to sleep because these my critical sections are likely to be very large they involved disk accesses milliseconds of time right. So, you want blocking locks alright.

So, one way to implement blocking locks is to just use a bit called busy bit in the buffer structure to indicate whether this bit this buffer is locked or not. So, right and use a buffer cache lock a global buffer cache lock to synchronize accesses to this busy bit per buffer right.

So, the idea is that I am going to take one global buffer cache lock and then to do mutual exclusion over these busy bits. And, then once I have locked so, these busy bits are going to act as per buffer locks and then once I have taken the lock then I am going to release the global buffer cache lock right. So, the global buffer cache lock is a spin lock that allows you to implement these blocking locks via these busy bits right.

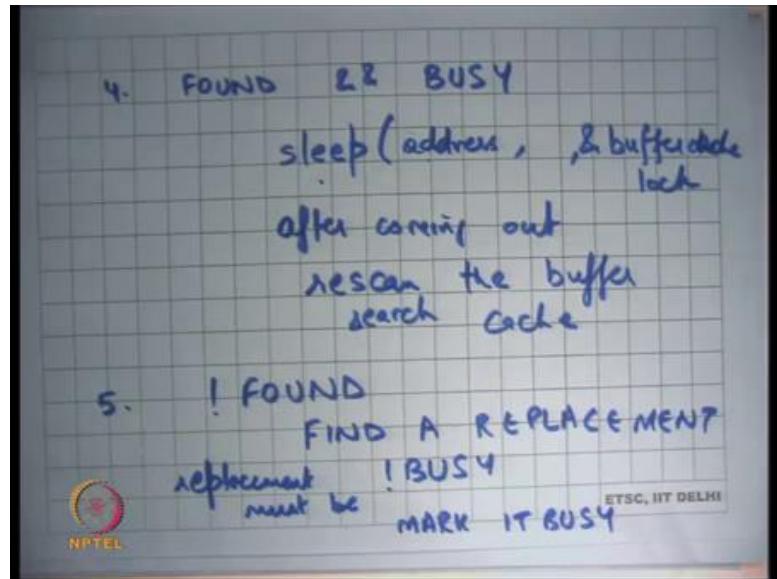
(Refer Slide Time: 18:43)



So, a typical operation would be that let us say I want to access a block. I will first acquire global buffer cache lock alright. Search if block in cache right. So, this is likely to be a fast operation. All you need to do is memory accesses to figure out whether the block that you are looking for is already in cache or not alright.

So, what are the; what are the few options? Let us say found and not busy right. This is the best case I found it in the cache, and it is not currently busy mark it busy right. Release buffer cache lock that is a global buffer cache lock and return the buffer right. So, notice I am using the buffer cache lock to provide mutual exclusion over my index structure and to provide mutual exclusion to accesses to the busy bit alright, that is all alright. So, now, what is the other case?

(Refer Slide Time: 20:25)



Let us say I found right. Let me write it on the different sheet let us say found and busy all right. So, I acquired the buffer cache lock I went through the index and I found it to be busy right. Recall that these busy bits are being mutually exclusive protected against a concurrent accesses by the buffer cache locks.

So, you have been you will either find it busy or you will not find it busy, it is not like their concurrent accesses happening to busy bits. So, it is found and busy and then what you need to do?

Student: Sleep.

You want to sleep right on xv6 you want to sleep that is block or sleep right. So, and you also want to sleep and the release the buffer cache lock right that is the second argument of the sleep. What is the first argument of sleep? You just need to provide some channel on which you are going to sleep.

So, let us say the sleep is let us say buffer address or address right. So, some channel which make sense in this case you basically saying I want to wait for this block to get freed. So, let us just wait on the channel which is then address of this block let us say this block address of this block.

And, and so, at some point, so, while you sleep you also release the buffer cache lock. So, now, other threads can be doing their operations on it and at some points somebody

is going to mark this particular buffer as not busy and it is that threads responsibility to call wake up on that particular address and so, I am going to come out of sleep. So, after coming out I can be sure that I have reacquired the buffer cache lock and what should I do this time?

Student: Again, you check.

Again, go to the step one and check if the buffer still exists right. The moment I release the buffer cache lock I have not I cannot assume anything about this state of the buffer cache of on reacquisition of the lock. It is possible that after I release the buffer cache lock somebody came in and not only marked it not busy, but also evicted it from the cache right.

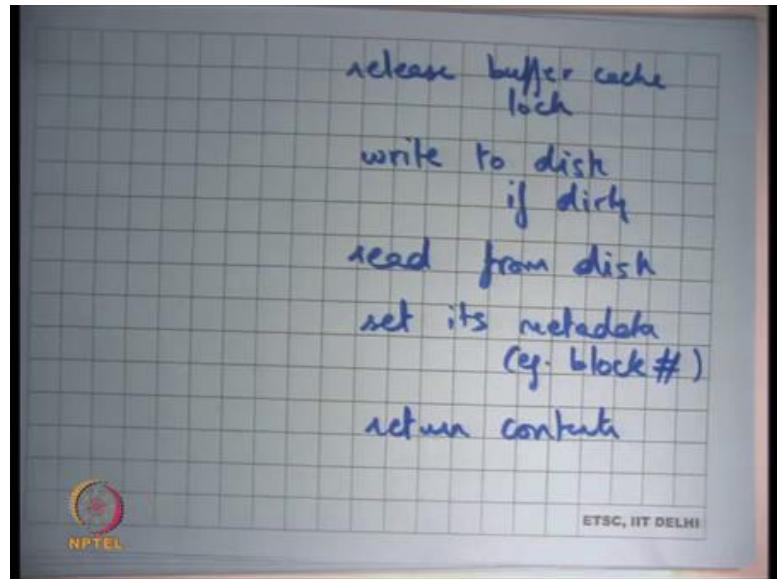
So, after coming out rescan the buffer cache or research alright and do the same thing. And, let us see what is the 5th case or what is the third case, not found in which case you will just read you will find a replacement buffer right.

So, once again you are going to find the replacement with the buffer cache lock held. So, you can be sure that there is not so, that is guaranteeing mutual exclusion. So, recall that all the in-memory data structures are being protected by this global buffer cache lock, but it is the individual buffers that are getting protected by fine grained locks. So, you find a replacement holding the buffer cache lock.

And, what so, firstly, you and make sure that the replacement should be not busy right. So, replacement must be not busy; if it is busy somebody is using that block so, I cannot just replace it right. So, the busy is also guaranteeing making sure that you know while you are doing something on while you are accessing a buffer while some other thread is accessing the buffer nobody else can actually evict it right. So, make sure that is not busy.

Once you have found the replacement you mark it busy. You mark it busy because now you have decided that you are going to operate on this particular block and what you are going to do.

(Refer Slide Time: 24:33)



So, once you have marked it busy at this point you can release the buffer cache lock right. So, basically it is you have to use the global lock to get the fine grained lock and then once you got the fine grained lock which basically means you marked it busy you release the global lock right. Now, you have the busy bit set so, nobody else can touch it till you make it not busy right.

And, now while you are holding the busy bit on that block you can write it to disk assuming it is dirty; so, if dirty read from disk the new block that you wanted to read you know set its meta data, example block number alright and return the contents right. So, I am using so, here is a very good example where fine-grained locking is absolutely necessary to have any kind of performance.

And, you are using a global lock to get these fine-grained locks. You do not want these fine grained locks to be spin locks you want these fine grained locks to be blocking locks, and so, to ensure to do that you need a global spin lock to synchronize accesses to these blocking locks and these fine grained locks are basically making sure that while you are working on something it does not get evicted nobody else touches it while you are working on it alright. So, but what is the problem with fine grained locks?

Student: Order.

There is one problem right we all know.

Student: Order.

Deadlocks, ordering. So, deadlocks. So, anytime you have fine grained locks there should be an alarm bell that you know I have to worry about deadlocks. So, what can what are some bad things that can happen? Now, let us just take this example. Let us say somebody is creating a file he allocates an inode.

So, he takes a lock on the block that contains the inode right and then he adds a directory entry record then he tries to take a lock on the directory entry record. So, first it takes a lock on the inode, then he takes a lock on the directory entry lock. If there is some other thread let us say unlink that first takes a lock on the directory entry record and then takes a lock on the inode then there is a possible deadlock right.

Student: (Refer Time: 27:12).

Well, I mean if you want this entire creation to be atomic you would want to hold both locks at the same time right. Otherwise what will happen is if you release the lock at this point; so, you allocate an inode and then you release the lock in the middle some other thread comes in so, it is going to see some inconsistency in your data structure, in your file structure right.

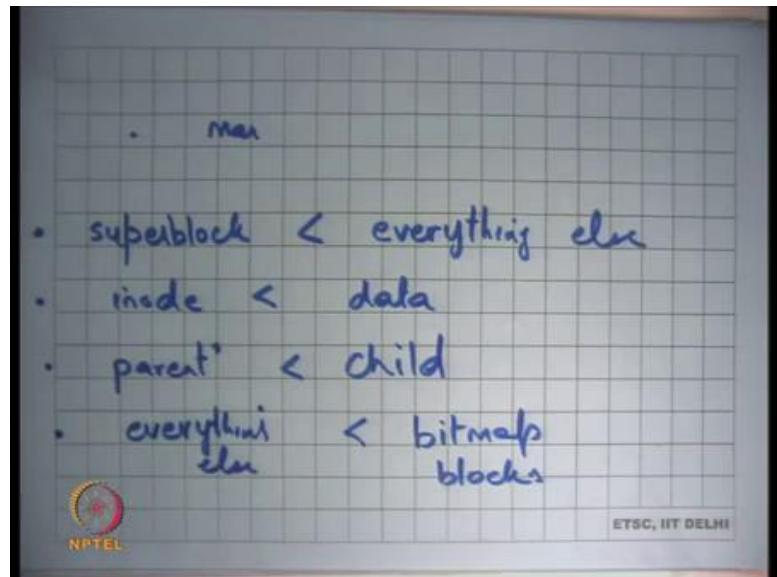
So, if you want that you know this entire creation. So, it is possible that the entire creation need not be atomic to ensure that inconsistency do not people no thread can see inconsistent data structure. So, there are some invariances are having to be maintained right. So, in general when you are talking about concurrent access and synchronization you basically first want to think about what are the invariants that need to be maintained.

And, the invariant typically looks like this: if I have been able to acquire the lock then I can assume that the data structure looks like this right. For example, if I have been able to acquire a lock on the tree then the tree must be a tree it should not have any cycles right or there should not be any dangling pointers and things like that right.

So, if you release the lock somewhere here then you are releasing the lock at an inconsistent state. So, the idea should be that you take a lock, and then you perform some operations and only release the lock when you leave the system in consistent state all

right. So, you take two locks here let us say to make sure that and so, they can be deadlocks right.

(Refer Slide Time: 28:55)



So, you need some kind of global orderings right. So, some orderings that that xv6 follows and the this is there is something that I was just looking through is that you know. If there is some operation that needs to touch the super block, then mark super block first right. So, let me just do ordering.

So, let us say super block before anything else before everything else right. So, this the less than sign basically says if you want this operation to do something on the super block sector then it should be taken first right ok, then inode before data right.

So, if you want to do something on the inode block and the corresponding data blocks, we will first take a block lock on the inode and then take a lock on the data and you the programmer needs to be careful about doing this right.

Let me look at some example. For example, here file unlink is an example where you first write to the data blocks and then you write to the inode, but locking should be in the opposite order alright otherwise you know you have deadlocks. So, programmer needs to be careful all right.

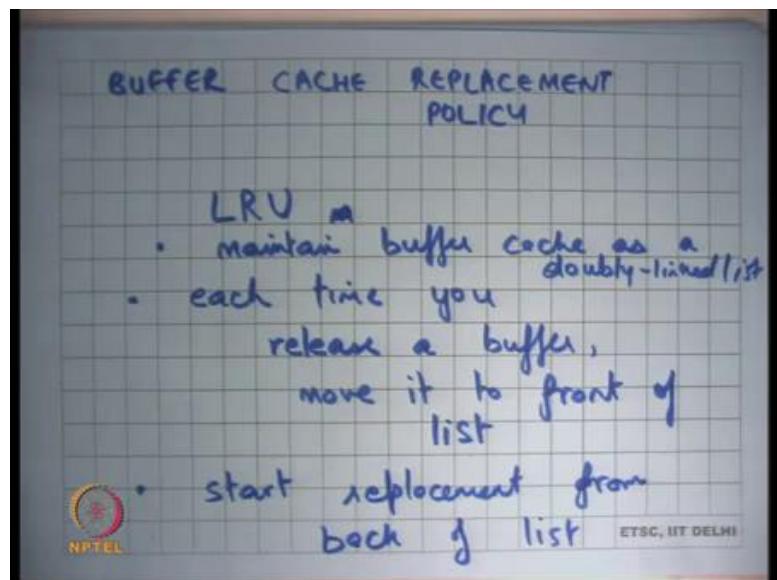
Then, parents parent before child, right. So, not only does do you have a hierarchy in the file system metadata, there are inodes then there are data blocks you also have a

hierarchy in the directory structure right. So, there are parents inodes and then there are child's inodes. So, parent directory inode and the child directories inodes; parent directories blocks, child directories blocks and so on right.

So, there needs to be some order on that as well. And, let us say everything so, bitmap blocks. So, nothing after bitmap blocks, so, everything else before bitmap blocks right. So, you should not hold a bitmap block and then try to hold a lock on something else right.

So, just a very good example of you know of a relatively complex system where you have fine grained locks and then you need to have some global order and there are lots of different types of entities and you need to have some global order on these different types of entities I mean. So, this is something I mean these are some invariants that xv6 could follows. You can check it for yourself you know if there are any if they are more, they are less let us find out let us know alright.

(Refer Slide Time: 31:41)



Finally, let us look at the cache replacement policy. So, recall that unlike the virtual memory subsystem, where you were really cared about the speed of the hit path right; the hit path was a memory access and you did not want anything any overhead on the hit path. The maximum you tolerated was setting up of a setting a bit, the access bit, and that too was done by the hardware. In case of file system, you have more flexibility.

Assuming that the file system is being accessed by a system calls like read and write, there is anyways a lot of overhead of taking a trap, making function calls, and potentially even making a disk access. So, the whole path of the hit path itself is quite expensive. So, you can do much more than just one bit for the access bit right.

So, in fact, the xv6 file system implements LRU for the buffer cache and this is the way it works. Each time you release a buffer so, maintain how is it implemented maintain buffer cache as a list as a doubly linked list all right and each time you instead of access I am going to say release a buffer.

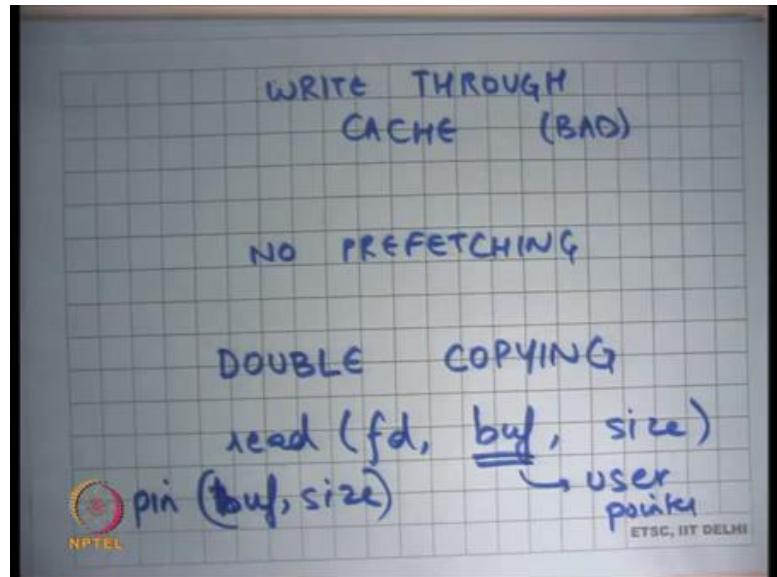
So, at the time of access you do you know when you first access the buffer you mark it busy. As long as it is busy it cannot be replaced, so, you do not need to do anything at the time of access. At the time of release that is the; that is the time you can basically say that I want to do something about this buffer to make sure that it gets released, it gets evicted the last because I have just accessed it. So, it is most recent access. So, it should be released the last.

So, each time you release a buffer move it to front of list all these doubly-linked list right and start anytime you want to replace you start replacement from back of list right and that is LRU for you right.

So, each time you access something you just move it to the front of list, you want to start replacement you start from the back of list. Recall that when you start replacement you look at the last you will look at the first you look at the last element and you get the first one starting at from the last that is not busy that is the one you will going to release. Anything that is busy is currently getting accessed.

So, the you know that is not a good candid for replacement any ways alright. So, that is LRU cache replacement for the buffer cache in xv6 alright.

(Refer Slide Time: 34:53)



Rather write through cache ok. So, write through cache is there for simplicity reasons, but let us look at is this the great policy. It is a very it is actually a poor policy from a performance standpoint each time you write something to a block it goes all the way to disk. Even if you overwrite some things a 100 times, there are 100 write through the disk. If it was write back policy, then you could have absorbed 99 of those writes in memory and just given 1 write to the disk right.

So, we all know that write back has this nice thing that it absorbs multiple writes to the same data block. More importantly given the physical characteristics of a magnetic disk write back cache is a much better option than write through cache right.

Write through cache basically means that the disk write is on the head path or on the critical path on the and so and you can only write you basically have only one outstanding IO or very small number of outstanding IOs under time. With write back cache you can batch many writes together and that way you can get much more efficient utilization of the disk bandwidth right.

Recall that elevator scheduler inside the disk controller so, if you give lots of things together that is the much better option right. So, disk a write through write back cache is better not just is even more relevant in the case of a magnetic disk because of because right batches are batching rights is a good optimization right.

Secondly, it has no prefetching let us see what it means. So, let us just say this is bad right a real file system would probably not want to write through cache it will want to write back cache and we will look at some issues there.

Secondly, the xv6 has no prefetching, what does it mean? Quite likely most many workloads have a very sequential behavior sequential varies. So, let us say I do I execute a grep command. Grep commands just going to go through the file in sequential order from byte 0 to byte last right.

And, so there is a very sequential thing let us say I implement I execute grep on top of the xv6 file system, what is going to happen is it is going to ask for block 0. The block 0 is going to get red and the by the time I actually start processing the block 0 the disk is already rotating. So, block one assuming that the disk was contiguously laid out on disk or the file was contiguously laid out on disk you are basically wasting rotations right.

If the disk if the file system could instead say I want block 0 through 10, then all those 0 through 10 could have been red in one go. But, now if you are going to say is 0, then 1, then 2, then each time you read a block you waste a full rotation before you get to one than you waste the full rotation you get to 2 and so on right. So, it is much more it is milli seconds more expensive than what could have been done if you had prefetching right. So, it is a so, that is the; that is the thing.

And, let us see there are actually double copies right. So, let us there is another fact about the file system that we have as we have seen it so far, that is double copying. What do I mean? If an application says I want to read sector x, then first the sector x gets copied from the disk device to the buffer cache.

The buffer cache is in kernel data structure, the user cannot see it directly and then from the buffer cache to the users address space right. So, recall that as user will probably want to say something like read fd, buf, size, right? And this buff will be a user pointer right.

To implement this basically the file system has to first read from disk to the buffer cache and then from buffer cache to the user address space. Is that a big problem? Well, assuming for something like a magnetic disk it is not a problem because the disk access

itself is the bigger bottleneck right the memory you are copying is not that big a bottleneck.

But, let us say you were running on a very fast network device right then these kind of double copying cause actually do become bottlenecks right. So, you know think about giga 10 gigabyte networking. So, if this file descriptor was not pointing to a file on disk it as pointing to some network device and you wanted to read data from the network device at a very fast speed, then your first copying it to the buffer cache and then copying it to the user space.

Could you have done better? Well, one way to do this is let us say you can say that when the user says read fd, buff I take this pointer and I pass it all the way to the disk driver right. Recall that the pointer that I give to the disk driver is that of the buffer cache and that is having this problem of double copying.

The other option could have been let us take the pointer from the user converted it into its physical address and then pass it all the way to the disk driver and so, the disk driver directly writes to the user mapped memory and then I just written and so, there is just one copy. The disk driver directly writes to the user memory. What are some things I need to be careful about? Recall that the virtual memories of system do swapping right. So, it can actually swap out data or swap out address space regions from the user space. So, I need we need to make sure that this region pointed to by buff till size cannot be swapped out.

So, I need to lock it in memory right in all you know the term used is basically pin it in memory right. So, you pin them in memory; you basically do not allow it to go through disk swap do not allow that to get to disk number 1.

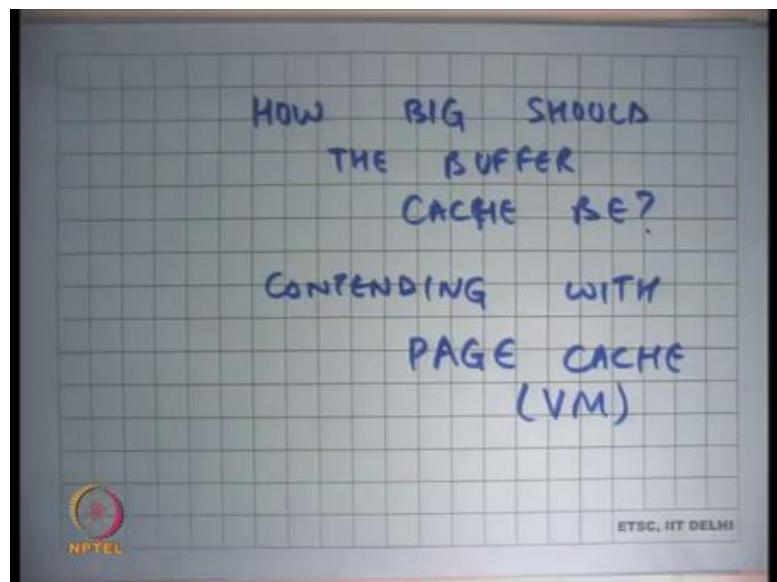
But, more importantly you disallow so, this read if there was locality of accesses by multiple processes then you also getting rid of you know you know basically also hampering locality. So, if this one process temporal locality is not getting properly address because buffer cache has it is nice property that it is a sheared cache between multiple processes.

Now, if you bypass the buffer cache and directly start reading to user address spaces if some other process also wants to read the same data then he will have to make another

disk access. So, you cannot reuse the cache right. So, in some situations you want to avoid double copying and these situations are situations where you care about very high IO bandwidths.

Think about network switches, network routers etcetera and you do not expect that much temporal locality of access across multiple processes right. On the other hands, there are other cases where you know double copying may be a reasonable overhead to take given its advantages alright.

(Refer Slide Time: 42:33)



And, finally how big should the buffer cache be? What is the tradeoff? Can I give the entire memory to the buffer cache what happens? I will have no memory for my virtual memory subsystem you know the process. So, recall that the memory is being used for two purposes as you have discussed it so far. One was to act as a cache for the virtual address space and another as a cache for the file system right.

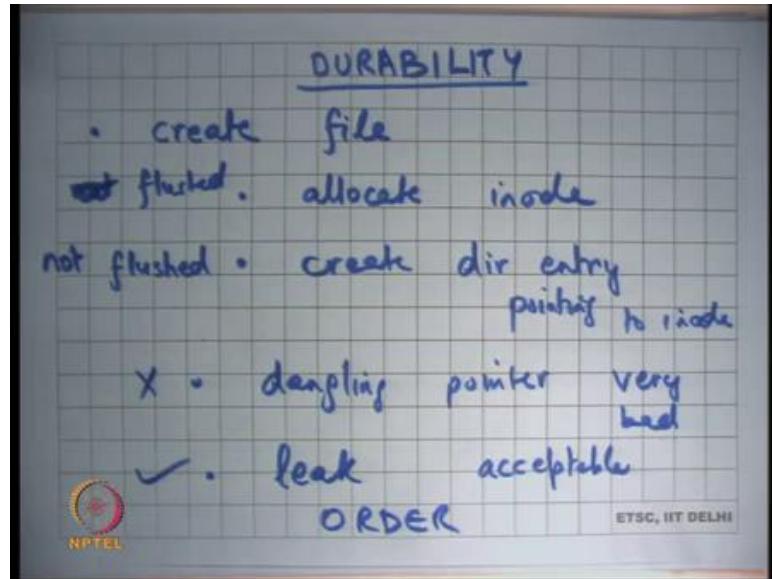
So, there are two caches in the memory one as a cache of the virtual memory address space and another as a cache to the file system and these are completely disjoint right. One is caching swap space data and memory mapped file data etcetera, and another is caching file data. And, both are completely different semantics also, and also different access patterns and replacement policies right and performance tradeoffs more importantly.

So, you are basically the so, the OS needs to decide how big should the buffer cache be and if it makes it big then there is little. So, it is contending with page cache right from the VM subsystem. And, so this decision can be made statistically. You can say I will have half of my memory for buffer cache and rest half for the VM page cache or you know, or you could do dynamic thing based on usage patterns.

So, basically you want to see if there is a lot of file system operations happening then you want to make the buffer cache larger. On the other hand, if there are not that many page faults happening so, you want to make buffer cache larger. On the other hand, if there are lots of page fault happening not that many buffer cache miss is happening then you want to make a VM cache larger.

And, you could do some coarse-grained dynamic tuning to figure this out right, but of course, xv6 just uses the static size (Refer Time: 44:53) ok.

(Refer Slide Time: 44:59)



Now, let us look at durability. So, one of the most important and interesting aspects of file system design is that state should not become inconsistent across power failures alright. So, let us see what can happen. Let us say I wanted to create a file and let us say I wanted to and to create a file I needed to make at least four disk writes and these are the four disk writes and at this point after I made the first disk write power caches, this is the power caches right.

Recall that all your busy bits etcetera was only in memory data structure and they were only there to protect against concurrent accesses by other threads. But now if there is a power failure then you have left the file system in an inconsistent state irrespective whether you use locks or not whether you use global locks or not, it does not matter.

The file system is still in an inconsistent state because then it comes back again you see that an inode has been allocated, but it is not mark pointing to anything right. worst ok. So, let us see what are some bad things that can happen. Let us say I am creating a file. There are basically you know at the coarse level there are two things I need to do. I need to allocate inode and I need to create directory entry right which points to inode.

Well, let us say a power failure happens and I have created the directory entry pointing to inode and I. So, I have let us say this has been flushed to disk and this has not been flushed to disk. So, I did I did both these operations, but I did this let us say first from a disk standpoint. So, I first created the directory entry and, now I was going to create allocate an inode.

But before I could allocate an inode the power went off. Power comes back again, what do you have? You have a directory entry pointing to some dangling inode, dangling pointer right. This is the bad thing. Why is it bad?

For example, let us say the inode that it is pointing to belong to some other user. So, now, I can potentially read the contents of the other users file. So, it is a security flaw from that standpoint right or it contains some garbage data which makes me feel that the file is actually really large right.

So, basically, I have a dangling pointer and the dangling pointer is pointing to some garbage and this garbage could be security sensitive data it could be something that come very badly confuse me right.

Let us see if the other thing happens. Let us say this gets flushed and this does not get flushed. So, firstly, you know. So, either of these things can happen. So, let us say I have allocated an inode all right, but I have not yet created an entry for it in the directory and the power goes down. Power comes back again what is the inconsistency in my system?

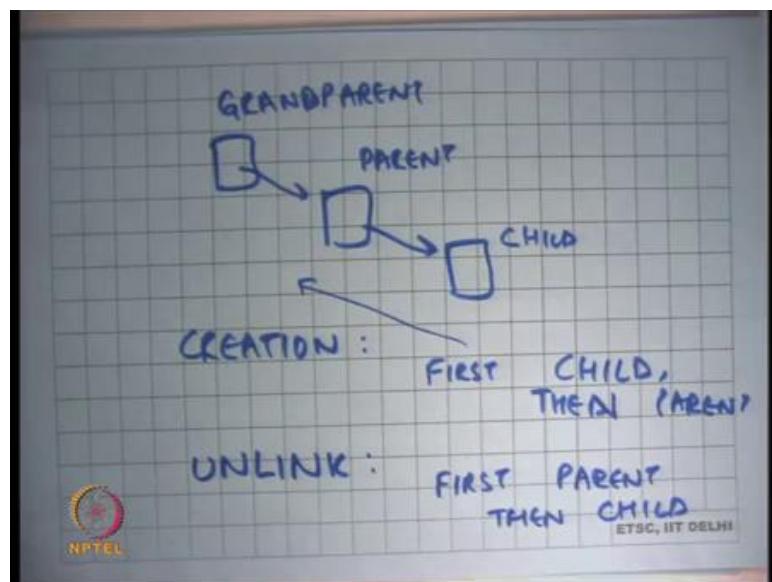
The inconsistency is that I have some node that is not free that is allocated, yet I have nobody pointing to it right. So, that is not as bad as a dangling pointer right. In the first case I had a dangling pointer and this we said was very bad and the in the other case we have a leak.

Why do I call it a leak? Basically, there is some data that is not usable anywhere or some blocks. So, the inode block is actually marked allocated, but it is not being actually be used in the file system and so, it is a leak. It is a space leak right. I will not be able to use it for anything else any longer right. So, this is acceptable.

Now, you have to so, one of these; one of these two things have to happen. So, to make this acceptable, one way to do this is to order the writes right. You say that whenever you create a file, I am going to first allocate an inode, flush it to disk and then create a directory entry.

If I follow obey this ordering then I can be sure then there will be never ending you know this is disallowed and this is possible, but this is acceptable. So, only acceptable things are possible alright. So, you order the writes to disk. Assuming let us assume a right through cache for the time beings. So, you would basically say that I want to create something I am going to first.

(Refer Slide Time: 50:13)



So, in general if there is a data structure that has something like this then at creation time. So, let us say this is you know all these are parent child relationships. So, this is let us say child, this is parent, and this is grandparent right. So, there is parent-child relationship between them.

In case of a directory and subdirectory it is a parent-child relationship between parent subdirectory and directory and subdirectory. In case of a file it is a there is a parent-child relationship between the inode and the data block; inode being the parent, data block being the child right. In the case of doubly indirect blocks, there is a 3-level hierarchy.

The top level inode, the indirect block and the data block and so on right. So, there is some parent-child hierarchy and at the creation at the creation time you are going to move in this direction right; first child, then parent right. So, that is going to ensure that you can only have leaks, will never have dangling pointers.

At the time of unlink or remove, what? In the opposite direction, first parent, then child right. So, if I want to remove something, let us say I wanted to remove this. So, I am going to first update the parent to say that you know there is no pointer here. So, I am going to first remove this pointer and at this point if there is a cache then I have a leak that child's data blocks are not getting are not accessible anymore, but there is no dangling pointer right. On the other hand, if I had first then a I had first an allocated a child then and there is a power cache after that then the parent would have had a dangling pointer alright.

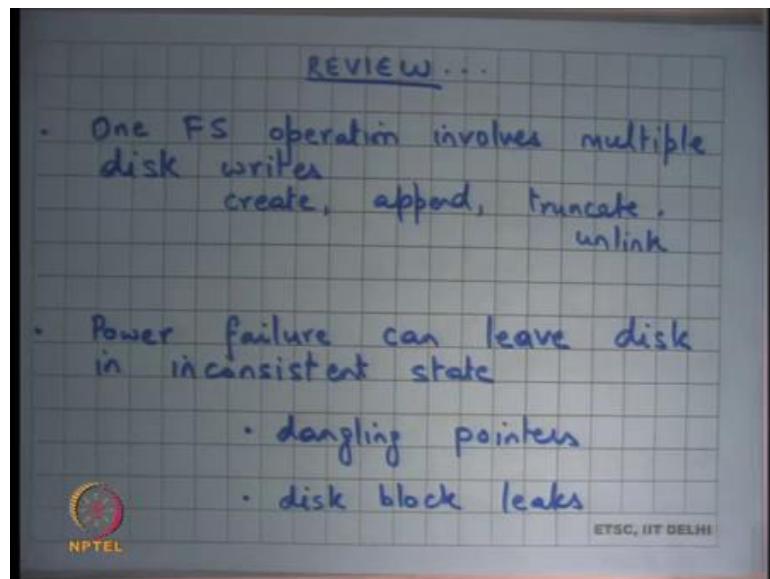
Let us continue this discussion on durability further in the next lecture.

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

Lecture – 34
Crash Recovery and Logging

Welcome to Operating Systems lecture 34.

(Refer Slide Time: 00:28)



So, we were discussing file systems and we saw that one file system operation typically involves multiple disk writes right. So, examples being create of a file you need to write to the parent directory, you need to write to the inode, you need to write to the data blocks right. Similarly append to a file you need to write to the data blocks, you need to write to the index, you need to write to the inode.

Truncate say same thing, but this time you are removing blocks from the file. Appending was adding files to the blocks to the file; truncate is just removing files blocks from the file. So, let us you truncate just removes the last few blocks from the file and unlink similar thing you just removing a file from a directory and once again all these operations will involve multiple disk writes typically 4 to 8 disk writes per operation.

And, the problem we were looking at last time was what happens if there is a power failure in the middle of one operation right and so the problem is that leaves the disk in

an inconsistent state and that can result in many bad things to happen. So, 2 and you can sort of categorize them into 2 types; One is dangling pointers where you have a directory pointing to a file which has not which did not get committed to disk which did not get read into disk and so the directory is pointing to some free disk block right.

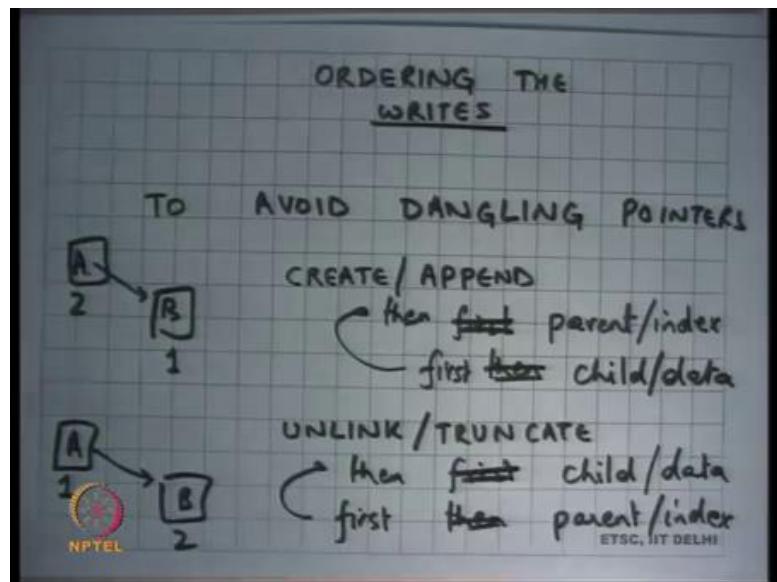
So, that is a possibility and that is a very dangerous thing. Firstly, because you know when the power comes back on there is no way to figure out whether this is a dangling pointer or whether this is a real pointer right. So, if that dangling pointer is actually pointing to some block that looks like an inode, suddenly the user will have access to this particular file, that it was not supposed to have access to right. Any logic that dependent on the contents of that file for the users program it is going to cause crashes more you know what worse that can happen is the user can gain access into somebody else's file right.

So, dangling pointers are a very bad thing and you would ideally want to avoid dangling pointers. The other problem that can happen is disk block leaks right. Here what can happen is that you basically initialize some data and you are about to create a pointer to this data from the index. Let us say you append to a file and you created some blocks at the end and now you wanted to change the index in the inode and make point to those blocks and there was power failure in the middle.

So, what happened is you have initialized some blocks you have removed those blocks from the free list, but you have not really created pointers to them before the power failure happened right. And, so when the power comes back again what you want to find out is that there are some blocks that are neither present in the free list and nor are they part of your directory tree or your file tree right.

And, so that is a bad thing also because there these disk blocks that will never be used and if you keep doing this, if you just keep having power failures unannounced power failures then eventually you will have lots of leaks lots of space in the disk that can remain unused forever. But, in any case disk block leaks are less dangerous than dangling pointers it is not a correctness problem it is a performance problem right. So, the solution you are looking at last time was let the file system order the rights in a way such that it avoids dangling pointers.

(Refer Slide Time: 04:12)



So, we say you know one of these has to happen you know after all there are multiple disk writes are happening and you know I cannot make them atomic with respect to power failures power can go get out any time. So, I have choice between dangling pointers and disk block leaks, and I will choose disk block leaks over dangling pointers.

And so let so basically, the idea is that I will order the rights to the disk such as the I will avoid the dangling pointers and which basically means that when you are adding something for example, I am creating a file I will first create the file or I will first create that initialize the data blocks and then create a pointer into the index right.

So, this way there will never be a dangling pointer because the index pointer will only be created after the disk data block has been initialized. In general, whenever there is a there is a data structure like this where block A is pointing to block B then you will first do this and then do this.

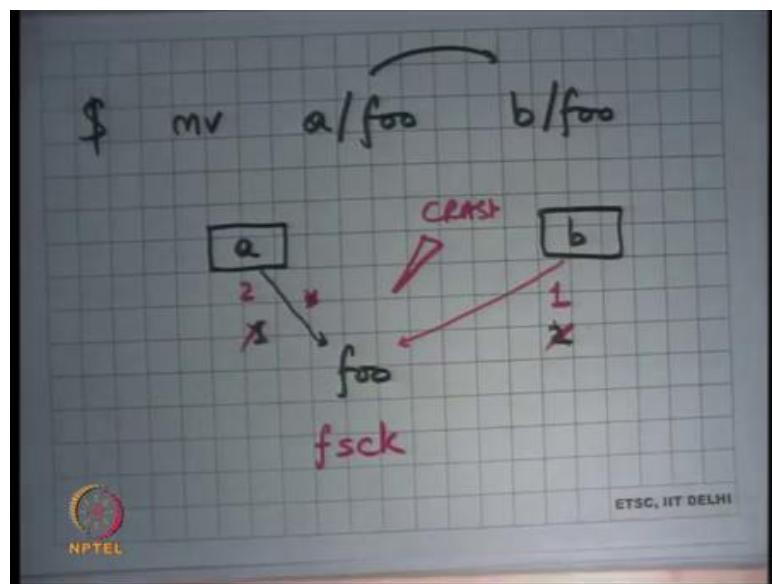
Student: Create an opportunity.

Because I am talking about create. So, you will first create the block B and then add an entry for it in the index A. So, you will first do this and then do this for create. On the other hand, when you were doing unlink for truncate when you are removing things you first want to remove it from the index and then deallocate or you know free or

uninitialized these blocks right. So, if I am doing unlink or truncate and I have a structure like this if I deallocate this first then I have a dangling pointer.

So, it is better to first write to A to remove that pointer and then deallocate B right. So, you will first do A and then B in this case right. So, first I have written it in the other way. So, yeah this is the mistake. So, first child or data and then parent or index and first parent and then child or data right; so, that is alright ok.

(Refer Slide Time: 06:25)



So, that is you know, but in all these discussion I am assuming that all operations have to happen synchronously which means I am not I am assuming the write through behavior right. I am saying that if I make an operation it should go to the disk because after all immediately because if I am writing my code I am saying update the index then update the data the file then and I wanted to happen synchronously because if it is not happening synchronously and if I am using a write back cache then even though I did these operations in a certain order what will really matter is in which order was the were these data blocks flushed to the disk right.

So, it can happen that in the cache I did this first and then this later, but when it was actually flushed to disk this happened before this right. So, all you know there is no use for ordering if there is a cache. So, in general you know doing sync writes is not very performant, but it has the nice safety property that you can order things too avoid

dangling pointers and this was indeed used for some years in many operating systems in the early days right.

But we know that this has a big performance penalty but let us also look at another thing is it always possible to do this ordering. So, let us say I had an operation which said move a file foo from directory a to directory b right. So, let us say I wanted to do this and so this is let us say the rename operation and I wanted this to be atomic with respect to power failures or I wanted to make sure that things are safe right.

So, here in this case there are two things that need to happen. Let us say this is a and this is b these are directories earlier a was pointing to foo and now you want to remove the pointer from foo and create a pointer to foo from here B right. So, now, in this case you may know you may want to say what should be done first should I remove this pointer first and then create this pointer or should I first create this pointer and then remove this pointer right. So, there are two options here I can either. So, I can either do this first and this second or I can do this first and this second.

In both cases my file system is on a reboot can appear in a very inconsistent state. So, it is no longer the case that one of them will only result in a leak if I do this first. So, let us say I do this first and the power failure happens before I had deleted the second pointer I have situation where one file is being pointed to by multiple directories right and this may not be acceptable and there is no way an operating. So, depending on the file system semantics they may or may not be acceptable. So, let us assume that it is not acceptable and there is no way that the file system can automatically figure out what to do right.

So, when it comes back again and it figures out there is something wrong there it will probably want to say I either want to remove this or I want to remove this, but has no way to figure out which one to remove. So, at that point you know there are couple of options either you ask the user look I see some inconsistency a file seems to be part of 2 directories or a data block seems to be part of 2 files you know it is similar then what you want to do.

So, those are options, and these are the things that you may have seen in programs like fsck right. So, there are these programs that run on you know if you have not mounted your disk cleanly then when you bring it back again there is a program that will do a global file system traversal and figure out if there are any inconsistencies and the

advantage that ordering gives you is that you have limited the inconsistency to only certain types.

In this case perhaps it makes more sense to do this first before that right because at least you are not losing data you have some inconsistency in your state, but at least you are not losing data if you had done if you had deleted first and then created then you would have lost some data right. So, you have to basically you have to do some ordering and depending on what ordering you are doing you have guaranteed now the kind of you have limited the types of inconsistencies that can happen on a power failure.

And, then you have a program that will that will be a long running program that will probably do a full file system traversal to figure out the inconsistencies and either fix them itself or ask the user to fix them for (Refer time: 11:21) yeah question.

Student: Sir, like in case the in case there is now in case this first and second both of you can (Refer time: 11:26) (Refer time: 11:27) and after that there was a power failure after that when the disk reboots how does it differentiate between whether or not the previous command was a mover whether it was something of the form of a short cut based or a.

Right. So, let us say I am let us say I am using this strategy were I am going to first create and then remove right and before I did a remove there was a power failure right. So, let us say there was a crash at this point and so I have a I have a file system state which has one file being pointed to by multiple directories. Now the question is how do I differentiate how do I know what was going on actually I have no idea right. So, there was no way to figure out what was going on at that time all you are going to do is you want to try to bring it in consistency state in some way.

Student: Sir no what I am saying is it might possible that this was a consistent state that there are 2 2 pointers to that like (Refer time: 12:15)

Yeah, I mean so your file system may actually; So, your file system may allow this kind of a thing and this may look like a consistent state in which case the file system may actually not throw any warning at all and it is for the user to actually now deal with things on its own right. But, let us I mean assuming that the file system does not allow this kind of thing and this is this is actually a inconsistency from in the file systems for semantics then you know it can throw warning to the user alright ok.

So, basically the idea that I have discussed so far is that you gone to order the rights on the disk in a certain way to limit the types of inconsistency that can happen and then you are going to run a program in case of an ungraceful shut down when you come back again that that is program is going to do a global scan and it is going to figure out what inconsistency they and trying to resolve it either automatically or using users help alright.

So, this is being used for a long time in lots of different operating systems and perhaps you have seen this also in the regular operating systems, but, but this has a few problems. Firstly, as a disk size becomes really large the time it takes to run this fs check program which is a global scan of the structure becomes very large right. So, just to give you an example you know a random read.

Student: Sir

Yes question.

Student: Sir, but earlier we have said that we have 2 copies of the table file table.

Ok.

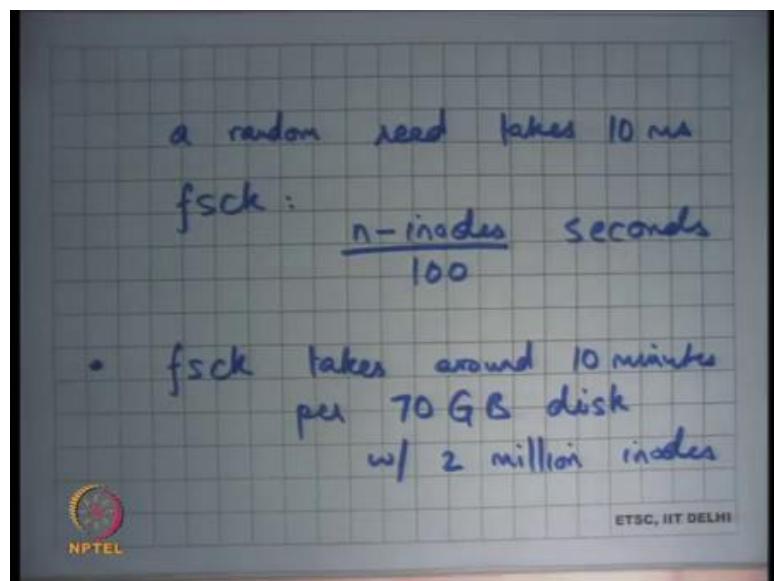
Student: File allocation table we said. So, cannot we use that to figure out what is that?

Alright. So, the question is that we have also said that for reliability we will duplicate state. Some state we will duplicate in the case of a file allocation table I will I will keep 2 tables right and so can that help in resolution of such conflicts see when we duplicate state we are basically doing it for reliability against stationary disk errors you know errors that can happen over years. And, things like that and you do not want duplicate state you know you would not deny a file system such that duplicate state basically involves updating twice for every update.

So, you know you would not design your file system typically to say that every time I make an update I will have 2 copies of every inode and every time I make an inode an update to the file I am going to write twice to 2 different inodes right. One could do that potentially and you know one of the solutions we are going to look at is similar in spirit, but, but that is not that is not a very performant design in the general case.

See basically whenever you are doing system design you basically want to say I want to speed up I want to make my general case as fast as possible and yet cover for the rear cases right you it is very it is a bad design to actually slow down your general case to take care of rear and if you are doing that then that is the you know that is basically an example of that alright.

(Refer Slide Time: 15:30)



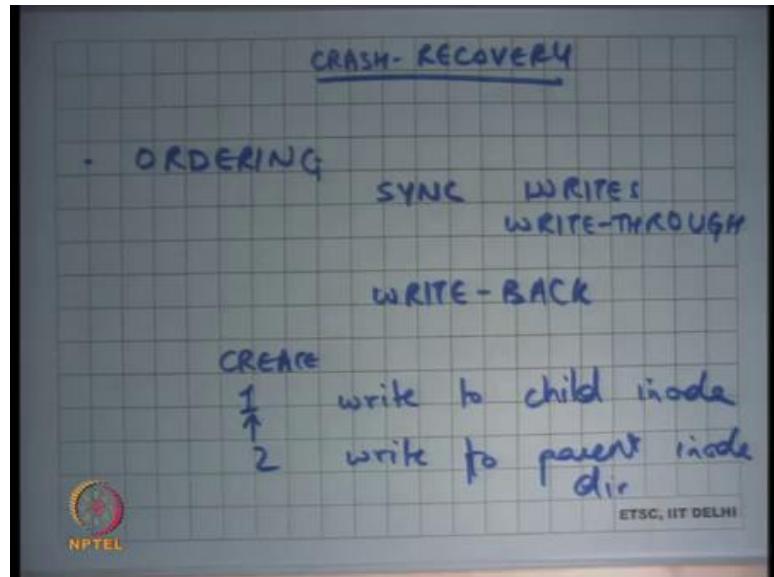
So, recall that a random read takes 10 milliseconds right and the. So, basically that means, that if I was to do fs check then the number of the time it will probably take me to do a full fs check would be somewhere like number of inodes which is representing of number of files in the system and let us say each inode is taking one random read.

So, you gone take n inodes by 100 seconds right; so, you can do a 100 inode reads per second. So, depending on the number of files you will do n inodes upon 100 seconds and that is really slow if you have thousands of files that is easily an hour write the as a data point fs check takes around 10 minutes per 70 GB disk with 2 million inodes alright.

So, clearly in doing so it is not it is instead doing it randomly read per inode the better thing to do would be if you are actually doing a full file system scan the better thing to do would be do an do an sequential read to read up all the inodes into memory and then do your traversal right, but even if you are doing that so this is the optimized statistic. So, even if you are doing that a disk which is roughly 100 gbs takes 10s of minutes right.

A disk that is terabytes will take hours and so on and so as the disk started to become larger and larger this idea of doing an fs check on a reboot started becoming less and less practical alright.

(Refer Slide Time: 17:21)



So, we so I am going discuss another method to be able to do this cache recovery, but before that let us also discuss. So, we have discussed one method which is ordering and with sync synchronous writes by synchronous writes I mean write through cache right.

So, whenever I write I write straight through disk there is no there is no write back going on. The other thing about synchronous write is it is very slow right. So, let us say I just image that you were to untar a tar file. So, and let us say the tar file has a 1,000 files inside it now just un taring a tar file requires a creation of 1,000 inodes and creation of each inode if it is a sync write it is going to take 10 milli seconds.

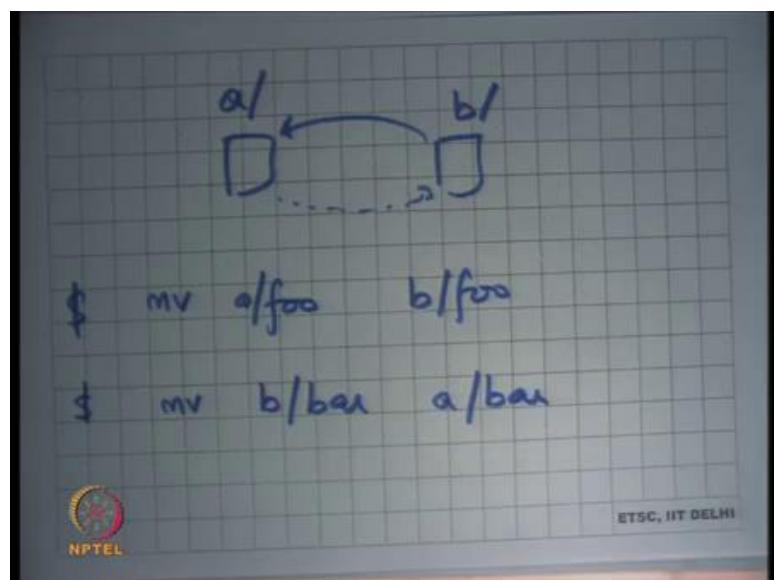
So, 1,000 nodes will take you know 10 seconds to create to untar one small a tar file which has a 1,000 files only right. So, sync write is not very practical. On the other hand, if you had a write back cache it would have done this whole untar operation in almost 0 milliseconds or less than you know less than a milli second basically.

So, you want write back and so the way this is done if you wanted to implement ordering with write back cache you would make the updates in the cache, but also store in the cache in memory ordering dependencies between disk blocks right. So, let us say I

wanted to do a create I will first write to child inode initialize it and then write to parent inode right parent directory inode let us say. So, let us say these are the 2 things had to do there few other things that that is to be done, but let us say just this 2 things to be done you have to write to the child inode and then you have to write to the parent inode.

So, what you will do is you will do these writes in the cache, but you will attach some ordering number to these. For example, you will say that this should be done there is an ordering dependency between the second block and the first block. So, the second block should be flushed only after the first block is flushed to disk. So, at the time of actually doing the replacement or flushing or writing it back in a bunch you are going to make sure that the things are ordered.

(Refer Slide Time: 19:55)



So, for example, you have multiple disk blocks let us say this is a's disk block and this is b's disk block and let us say a and b are directories then if I say move foo a/foo to b/foo. And, I want to make sure that creation happens before unlink then I will say that I will draw an edge from b to a saying that b should be flushed before a is flushed right. Creation should happen before unlink right.

On the other hand, let us say after that somebody executed a command called mv b slash bar to a slash bar. So, somebody create executed yet another command and so this time you wanted to draw an edge like this right this time you are moving in the opposite direction. So, you basically want to create you going to first create a link in bar and then

remove the link from in b, in a and then remove the link from b right. This time you basically want to say that this should happen before this right and here is an example where this dependency graph could have a cycle.

And, if it has a cycle then at a time of flushing it back to disk it is unclear which one you should flush first right. Let us say you flushed a first then you have lost it is possible that crash that happens after you flushed a then you may have lost the link to foo write. So, foos contents may have been lost. On the other hand, if you flush b first then bars contents could have been lost ok. So, that is cycles now how do you resolve something like this.

Student: In data driven what is to be updated in a and what is to be updated in as in not just the whole of data of a and then whole of data.

Okay so here is the suggestion I am I am maintaining ordering at block (Refer time: 12:54) I am saying this block should be of committed to disk or written to disk before that block I am without going into semantics of what is inside the contents of the block. And, so suggestion is instead of saying that this block should be committed before that is that block say these contents in this block should be committed between before that content those contents in that block.

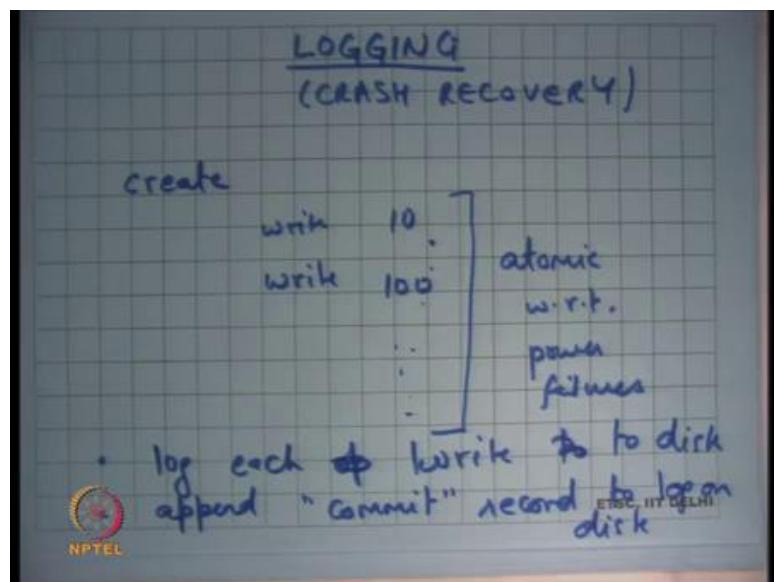
That is not a practical solution right because that is there is too much semantic information that needs to be stored and that it is basically almost like writing you know at the at flushing time you have to basically look at this semantics what is written at what bytes that has been written it is a directory or a etcetera I mean these are the kind of things you do not want to worry about at flushing time. All you want to care about is here some disk blocks that are in the in the cache they are dirty they need to be written back to disk.

What is the other thing that you can do? Well here is here is one suggestion. When this operation gets executed in memory the OS figures out that hey there is a this like this is going to be a cycle in your dependency graph. So, when you have when you see that there is a possibility of a cycle you stop that operation and you flush this disk blocks to disk such that all the such that this disk this edge gets removed.

So, once you flush them to disk the old edges will get removed which means you know the old state will get consistent and now you can perform this new thing right. So, every time you are making an operation you check the dependency graph if you see a cycle you hold on you flush the blocks that are involved to the disk. So, that the old edges get removed and so the new edges can get created without having a cycle in your dependency graph ok.

So, ordering with a write back and coupled with an fs check program has had been a has been a popular solution from a long time.

(Refer Slide Time: 23:51)



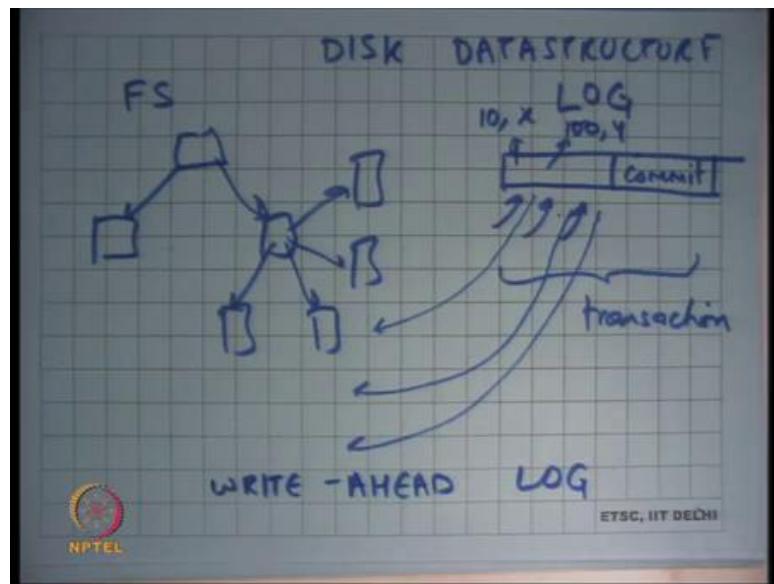
But with growing disk sizes it has become relatively unpopular and the other way to do cache recovery is login. So, let us see how login works. So, the idea is that let us say there is some system call let us say there is create hence going to do write to block 10 then 100 and so on right. And, then and that is it and want to make sure that these operations are atomic with respect to power failures ok. So, there are some operation that is gone to have multiple disk writes and then you want to have them atomic with respect to power failures.

Here is one way you could do it. Each time you see something like this you basically do not write the disk blocks to the file system at the time at this point you keep recording these operations in a log. So, basically start logging you maintain a separate data structure called the log on disk and each time you want to log write something you

basically say you basically log these operations. So, you say that I want to write to block 10 with these contents and you put that in the log. When you put it in the log you basically saying you're you basically indicating your intention that you want to write to this disk block you are not actually written it to the disk block right.

So, you log each operation each log is write I should say to log alright to disk. After you have done after you have done logging each of these writes you append a commit record to log on disk right. So, you basically first write that I want to write to all these blocks and then append a commit record to the log on disk alright and your operation is complete only after the commit record has been written. If there is a power failure before the commit record has been written it is as though none of these writes have happened. If there is a power failure after the commit record has been written it is as though all these writes have happened.

(Refer Slide Time: 26:34)



So, let us see I have a disk let us say I was to draw the data disk data structure. So, I have some tree like structures you know different types of tree; one is a directory tree and other is an inode tree and so on. So, they are tree like structures on the disks. So, this is let me call this is the file system and then there is a sequential structure which I call the log.

Each time I want to make an operation I basically start writing to the log and then I write a commit recall that we made the assumption that the write of a sector is atomic to the

disk. So, either the entire commit will be written or none of the commit will be written it is not like half of the record can get.

So, this operation is atomic and you using this operation to basically make the entire operation the entire sequence of writes atomic and after you have written the commit log commit block you are going to asynchronously make these writes to the file system alright. It is after you have written the commit record.

So, let us see once again. I wanted to make an atomic operation which involved multiple disk writes I will create a new transaction. So, this entire thing can be called a transaction. So, I will create a new transaction I will log the blocks that I need to write and log them to the log and then I will write a commit record to the log.

After that I am done as far as I am concerned the disk block the disk is not in consistent state and asynchronously I am going to flush I am going to copy the blocks in the log to their respective positions on the file system right. Recall, that all these logs have annotation saying block 10 contains x block 100 contains y and so on. And, so asynchronously I am going to write these blocks from log to the file system.

Now, let us see what happens if there is if there is a crash if there is a crash before the commit record was written no problem it is as though nothing happened right. If there is a crash after the commit record has been written, but before you have started applying the changes to the disk no problem the operation has finished atomically and now you can now do the same thing.

So, at recovery you can just apply the log to the file system. What happens if the crash happens in the middle of application of this log to the file system? So, let us say you know I have I have written the commit record after that I was asynchronously writing block 10 and before I could write block 100 there is a power failure no problem.

Student: (Refer time: 29:30) 10 we will do.

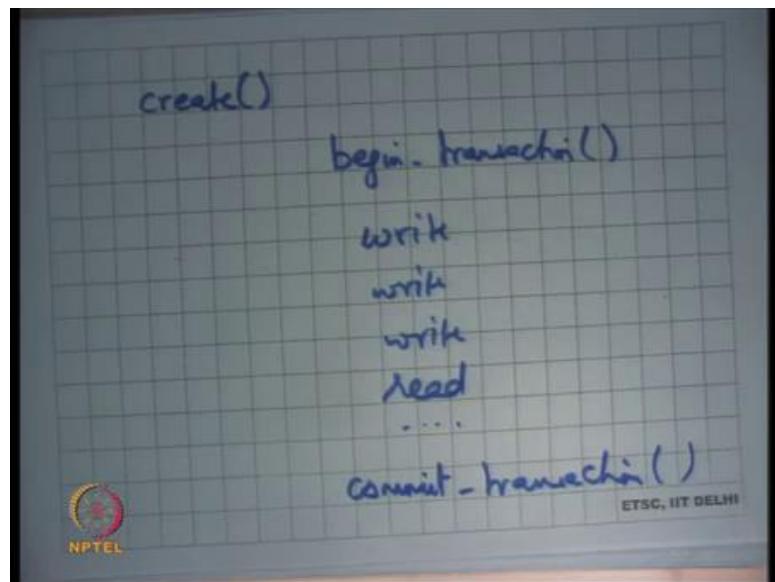
At recovery time you will again write ten, but with the same contents. So, there is no problem right. It is a you will just overwrite the 10 again, but with the same contents that is no problem either right. So, this is nothing, but a write ahead log where whatever you want to write you write it to the log first and you keep doing this and then you write a

commit record after that you actually push all those writes asynchronously to the real file system. So, we are all convinced that this would ensure consistent state even across random crashes on the in terms of power failures.

Notice that the write of the commit record on the log is acting as a serialization point. If a power failure happens before the commit record it is as though the transaction did not happen at all. If the cache happens after the commit writing the commit record as though the transaction happened in completion.

Even if there is the crash that is happening in the middle of your application of the log to the file system it is still consistent write this data structure of the file system plus log will always remain in a consistent state. There is no inconsistency that may happen alright. So, how does one write code?

(Refer Slide Time: 31:11)



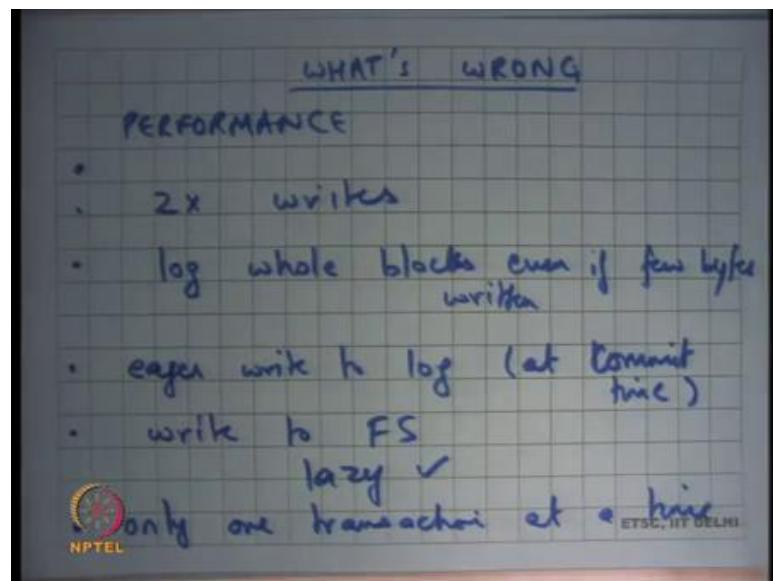
To do this let us say I have a cis call like create. I will just say begin transaction then you know write, write read etcetera and then I will say commit transaction. So, all I have done is that all the all the operations that I wanted to make atomic with respect to power failures I have bracketed them with begin and commit transaction calls right just like very similar to your locking acquire and release right.

And so, begin transaction is going to write to the log that I am starting a transaction. All these operations are going to append records to the log and then commit transaction is

going to write the commit record on the log and there is going to be another thread that is asynchronously going to apply the log to your file system alright.

So, it is very easy to write to basically there is the code structure does not change much you just have to enclose things that you want to make atomic with begin transaction and (Refer time: 32:25) append transaction. After you are done applying the log applying the transaction to the file system you can delete the log you can delete the transaction from the log so that it can get reused right. So, till it has not been applied to the file system you have to keep it around, but after you applied it to the file system you can now free it for use for the next operation alright.

(Refer Slide Time: 32:53)



So, what is wrong or what do not we like about this performance right what are something that are that are happening. Firstly, everything that I need to write I need to write twice I need to write first to the log and then to the file system. So, every operation basically has a 2x overhead in some sense right. So, 2x writes ok.

The second thing is I log whole blocks even if few bytes written. So, even if I just update one byte in the block, I have a log the entire block in my log. So, my log space overhead is larger than then what you would have wanted it to be. Eager write to log. I am very eager to write to the log every time I do a write I basically immediately want to write to the log right. And, as soon as I do a commit, I want to basically make sure that the

commit record has been written to the log and then later I am going to write it to the file system.

So, is this avoidable? Can we not have eager write to log to the log. Well we will see this in the next lecture, but as it stands now this thing has a very bad performance each time I want to make a write I actually need to go to the log and makes this write. One simple optimization could be you do not do the eager writes to the log as they are being done. You wait for the commit transaction to happen and then all these 4 or 5 records that you written can be written in one batch to the transaction to the log right.

This is a small optimization, but this is still not good enough why is this not good enough I still need to know do this eager write on every commit. So, there is no not write back in the true sense every operation needs to be synchronously committed every atomic operation. I would have wanted that even atomic operations can be written back using a write back cache right. I want my write back cache to have more freedom in or more batching than just 4 or 5 disk operations alright then.

Student: Sir.

Write to file system. So, write to file system as we discussed can be done can be done lazily. So, this is this is write. Here we are saying that once you written it to the a log you basically lazily apply the log to the file system, and you can batch it in a and that gets a lot of performance right. Ok, there is a question

Student: Sir, suppose if we update a suppose if we update the log after fs 4 5 operations and whenever there is a commit sir, but in that case and before that we are just writing to a cache now, but suppose if there is a power failure while I am writing to the log from the cache and the commit keyword is written, but other operations are not written.

Ok so, good question I am saying you know there is an optimization that a transaction need not be written to disk or their blocks in a transaction need to not be written to disk eagerly you can know keep them in cache and then at commit time write the entire log transaction to the log in one go and the question is in when you are writing it back if the commit happens before if the commit is written before the other blocks then there is a problem right.

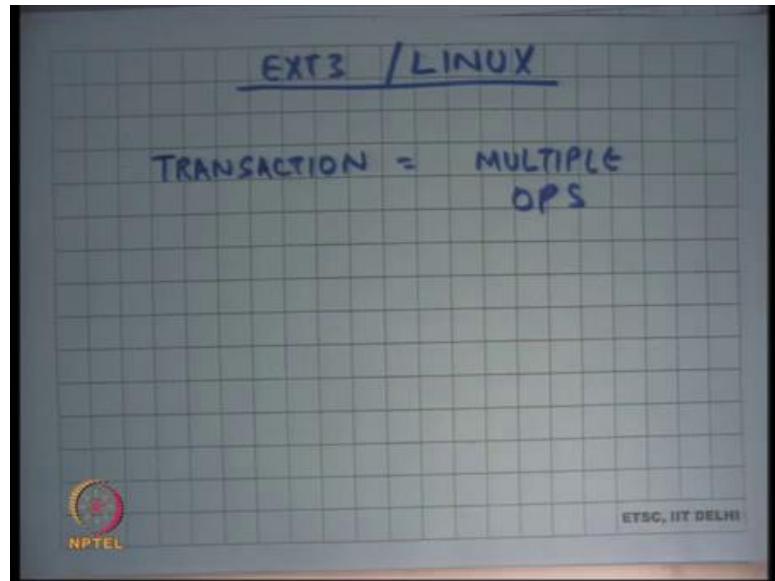
So, at the time of flushing it you have to basically make sure that you know there is some ordering in which you are done doing it typically the way this will be done is that you will issue all the transaction records in one go. So, all the transaction records happen in one go and then the commit record happens in a second iteration. So, in that way you have 2 sequential writes to the disk right.

So, you waste one full rotation assuming that the log is written sequentially you first make one sequential write to write the entire transactions blocks and then you make one sequential write to write the commit and just to make the ordering you have to do 2 instead of one. So, that the disk does not reorder them internally. Modern disk interfaces allow you to specify that you know here are 5 reads 5 writes, but make sure that this 6th write is after this all these 5. So, you know that way you can even avoid this extra overhead of multiple writes good.

So, we have seen we have seen logging in it is very raw form where we are basically, we are using the log we are eagerly writing to log we are logging the whole blocks and eagerly in the sense at commit time. Also, we are making another problem with this thing is that only one transaction at a time right. Why is it is true. Let us see let us say I have this code begin transaction and then I start writing something then I commit transaction to ensure atomicity across between multiple access to the file system the way I have discussed it so far you basically want to make sure that only one of them one transaction is active at any time.

If the multiple transactions active at any time, then they are more problems to be dealt with right. So, the way we are discussed it so far there is only one transaction that can be active at any time and that in itself is a very big performance problem because if there are you know lots of users running lots of programs to completely different parts of the file system they get serialized because of this common log across the entire file system right; so, not a good idea at all alright.

(Refer Slide Time: 39:28)

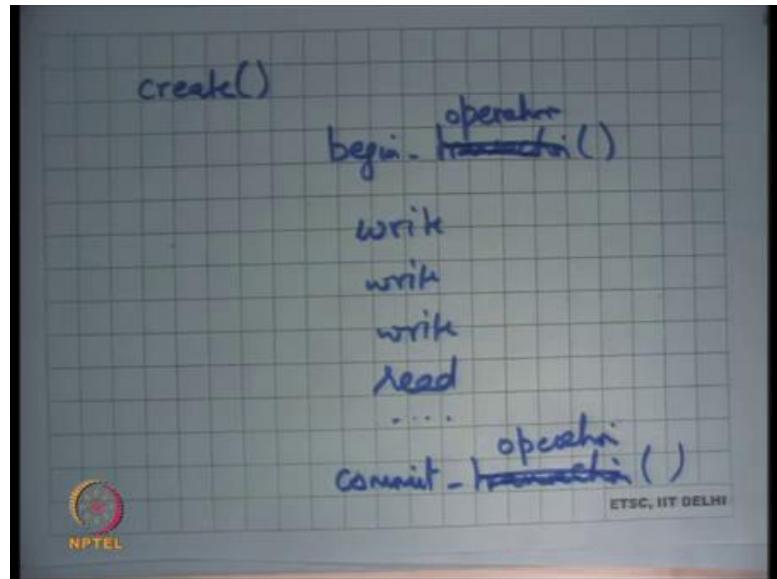


So, let us see how we can fix it and I am going to look at the ext 3 file system on Linux right. So, the ext 2 file system was basically based on the ordering and fx check program that we discussed earlier, but the ext 3 file system introduced logging and, but in an efficient way and we gone see exactly how this is done. So, we saw some problems with the way that we have discussed so far and let us look at what can be done. Firstly, a transaction is not one operation, but multiple operations right. So, if there are 100 creates happening at the same time, they are all made part of one transaction.

So, one; so, a transaction basically represents locality in time. So, basically, we say I start a transaction and the all the file writes that are happening now will belong to this transaction. And, then at some point I am going to say stop transaction when I say stop transaction all the operation that are completed belong to this transaction and all the transaction that are ongoing. I will wait for them to complete. And, when they complete all these operations together form one atomic unit right and this atomic unit gets flushed to disk in a log.

And, then there is one commit record for this entire big chunk of writes possibly by multiple users multiple processes different parts of the file system completely different operations right. So, what you are doing is you are making you are clubbing lots of different atomic operations into one large transaction.

(Refer Slide Time: 41:25)



So, basically what this means is that these begin transactions and the commit transactions can be replaced by begin operation and commit operation. And, it is not necessary that at the time of committing the operation you actually commit the transaction on disk you just basically say that the operation is finished. And, so the invariant is that a transaction will have either the entire operation in it or none of the operation in it and a transaction will never half of an operation and so you basically club lots of different operations into one transaction and so that gets rid of some problems.

So, for example, only one transaction at a time you have solved it right. So, you can have lots of different operations at the same time and then you can choose to commit them at your own will. How do you choose when to commit a transaction? You can say every 5 seconds every 30 seconds completely reasonable choice right.

So, for example, you know modern operating systems do not give you any guarantee about if you are writing something whether it is actually persistent on disk or not, but every 30 seconds let us say the transaction will get closed. And, so because the transaction got closed it will get written on to the disk and at that point you can be sure that the data you wrote 30 seconds ago is very likely on the disk now right.

So, this interval; so, you can just choose to close the transaction at will whenever you like and when you close the transaction at that point you make sure that all the operations

that started before the close of the transaction will get committed on disk right. So, you fix this you also fixed eager write to log.

So, now you can because you are using a transaction worth of thirty seconds of writes you can actually batch all of them and write them all together and. So, it is not an eager write of log to log you actually instead of writing 6 operations you are writing 6,000 operations together to the disk. So, that is much more efficient right.

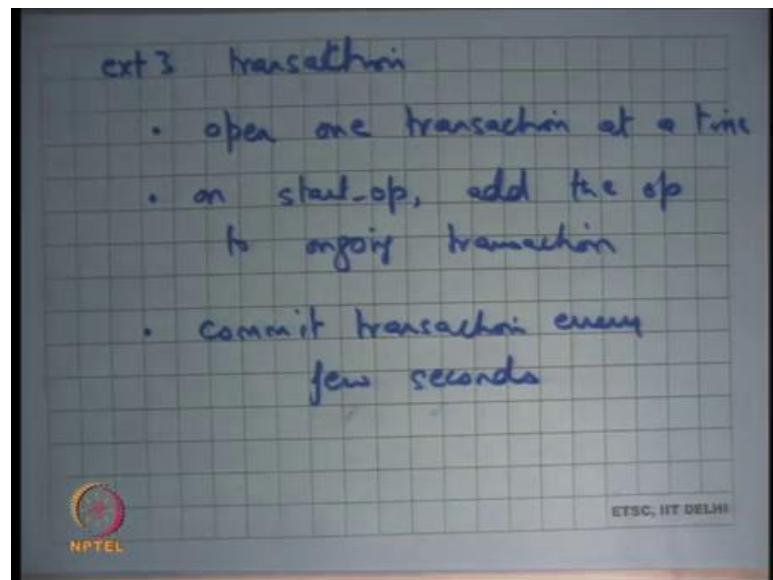
So, this one is already good write to fs is lazy log whole blocks even if few bytes are written. So, there is still log whole blocks even if few bytes are written because the you know it is extremely hard to keep track of which bytes have been written and which not. So, you log the whole blocks, but the nice thing is because there you are you are making out transaction. So, big if there are multiple operations that wrote to the same block. So, you log the whole blocks even in ext 3.

But the nice thing is because you are doing lots of different operations batching into one transaction you do not have to these blocks may have been modified many times within the same transaction. And, you do not need to record all those blocks all those different versions of the blocks you only need to log the final version of the block.

So, let us say if this block was modified a thousand times in this thirty second interval all you need to log to disk is the last value. So, you still log the whole block, but you just absorb lots of writes and doing so this also becomes better alright and 2x writes remains, but the nice thing is because you are batching. So, many writes together in log right it is all it is complete sequential write to the log. So, that is fast as we know there is only one seek and one rotation and you would do write at 50 to 100 megabytes per second to the logs. So, that is fast.

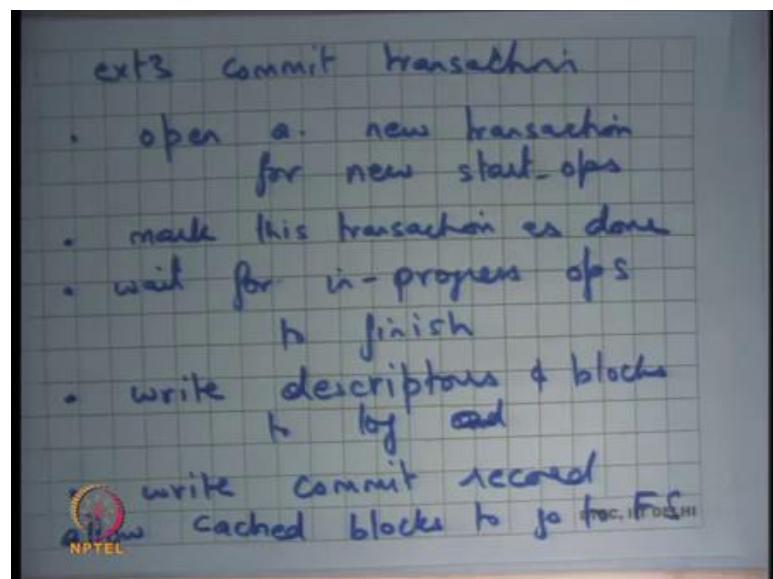
Also, now all these writes now need to be applied to file system, but that is a huge transaction. So, because it is a large transaction you can have lots of IOs and flights simultaneously. So, writing to the disk is also faster you have much better disk bandwidth utilization. So, essentially the larger the transaction the less the 2x write problem is right and because I have made my transaction as large as I want there is no problem right.

(Refer Slide Time: 45:29)



So, let us look at how ext3 works. So, let us look at an ext3 transaction. Open one transaction at a time on start op add the op to ongoing transaction alright. This is all in memory operations. Each time somebody starts an op you basically add the op to the ongoing transaction and commit transaction every few seconds or few tens of seconds depending on what you want how much reliability and performance trade off you want.

(Refer Slide Time: 46:30)



Let us see how would you commit transaction. Firstly, open a new transaction. So, when you are committing a transaction you basically say that all new ops will now belong to

the next transaction. You basically first close your current transaction which basically means any operations that are happening after this point will now belong to the next transaction. Mark this transaction as done. Wait for in progress ops to finish.

So, there could be some ops that are started, but are not yet committed or haven't stopped finished right. When you close the transaction there are some partially completed ops you just want to wait for those ops to finish before you actually write the committed record right. So, that is how you basically maintaining atomicity right. You have basically said that I am I can close the transaction at any point you want, but when you close it you also wait for all ongoing or all operations that have started before this those to finish. So, that is how you basically ensure atomicity of each operation.

Then write descriptors. So, descriptors are basically these blocks which say which contain meta information about which block and etcetera which block which version number etcetera. And, then what is sort of data right descriptors and blocks to log and wait right and then write the commit record and that is it right that is your finish, that is your commit transaction and then asynchronously you are going to write the contents or whatever the log is saying to the file system right allow. So, after you written the commit block you can now allow the blocks that have whose contents have been logged in the transaction to now get written to the actual file system alright.

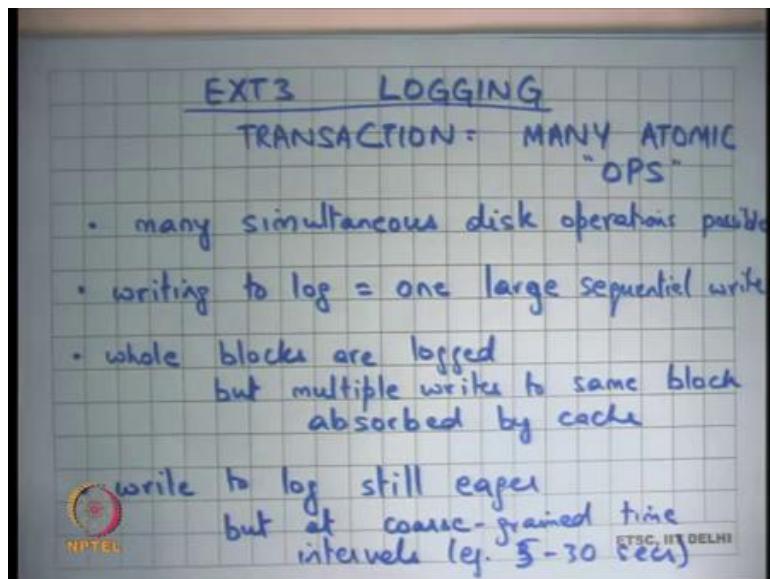
Let us stop here and let us discuss the ext3 file system in more detail next time.

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

Lecture – 35
Logging in Linux ext3 filesystem

Welcome to Operating Systems lecture 35.

(Refer Slide Time: 00:31)



So, we are discussing about logging as a way to implement fast cache recovery, we first looked that very simple way of doing logging where every disk operation that needs to be a atomic was considered a single transaction. And, there was a commit after every disk operation right and we said that there are lots of problem with that.

And then we said, but you know in practice you would extend this idea and we were taking the example of the ext3 file system on Linux, where a transaction is lots of atomic operations bunch together into a single transaction.

So, in other words with respect to power failures either all these disk operations happen or none of them happen at ones. So, it is making a one big atomic operation out of lots of smaller atomic operations. And if you do that then firstly, you; so, the way firstly, in the previous case we said that only one transaction is possible at a time.

So, even now there is only one transaction possible at a time, but because one transaction can have lots of different disk operation many simultaneous disk operation is possible at the same time right. So, transaction is no longer causing serialization across multiple disk operations where disk operations are completed two different parts of the file system.

Of course, if the disk operation that was the same part of the file system then there is in memory locking to ensure that there is no concurrency problem, that we have discussed already right. So, there is buffer level locking to ensure that there is no; there is no atomicity violation by concurrent access to the same block. But here we are talking about atomicity with respect to power failures right and for atomicity with respect to power failures we wanted to have a transaction and we wanted and we could only have one transaction at a time.

And so, that was causing serialization, but because you have multiple operations in a single transaction that serialization problem goes away. Also writing to log is just one large sequential write and we know that sequential writes are fast assuming that you are doing you commit. So, the weight works as you open a transaction and you close it after every periodic interval.

Let us say every 5 seconds or every 30 seconds depending on what kind of guarantees you want. And, after interval you going to commit the entire transaction includes which include all the disk operations that have disk writes operation that happened in during that time. And so, one last sequential write is fast, whole blocks are logged still.

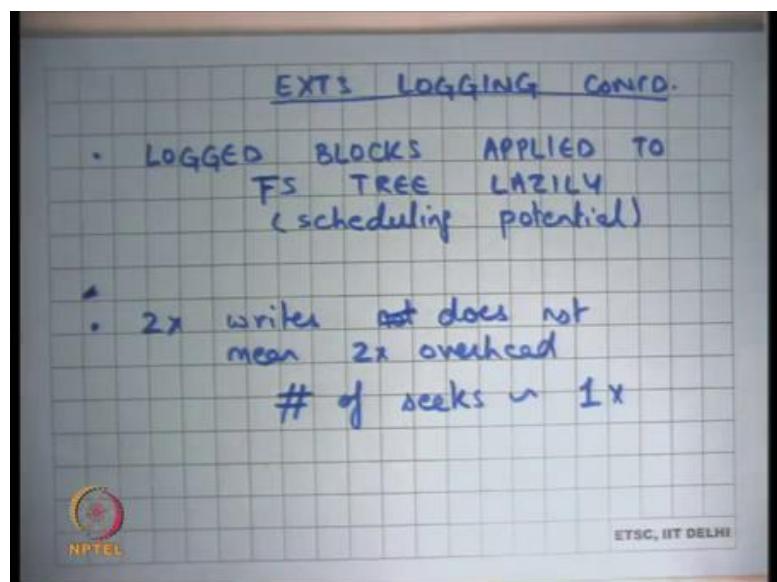
So, we said one of the problems with this kind of logging is the whole block is log, if you even modify 1 byte in the block the entire block get needs to be log into the log. And that is more space overhead in the log, but it is still present in ext3. The only saving grace is that if there are multiple rights to the block which is a very common case you know this likely if you for example, creating lots of files in a single directory then you probably making lots of writes to the same directory inode and you know 100s of writes to same like an inode.

You do not need to log all those 100 versions of that block; you only need to log the last version of the log right. So, in multiple writes to the same log get absorbed by the cache, write to log is still eager. So, when you commit you have to commit you have to write

the entire log at that time, you cannot lazily say I am going to commit it later because otherwise you completely lose your guarantees with respect to power failures.

So, it's still eager, but the nice thing is that these eager writes to the log are happening at very coarse in time intervals; 5 second or 30 seconds depending on what you have chosen. And so, it's not that big of a problem anymore alright.

(Refer Slide Time: 04:00)



Finally, log blocks are applied to the file system tree lazily. So, the commit of the log or commit of the transaction is eager, but the application of the log of the transaction blocks to the filesystem tree is lazy right. So, you can just do it whenever you know whenever you find time or whenever you find the disk to be idle or more importantly you know you can take a lot of logs and try to write them simultaneously.

And so, that gives the disk a lot of scheduling potential. So, recall that if you do lots of in-flight IO simultaneously, the disk can schedule at much better than if you have one IO time. So, you can and all these can be done simultaneously; so, have a lot of scheduling potential.

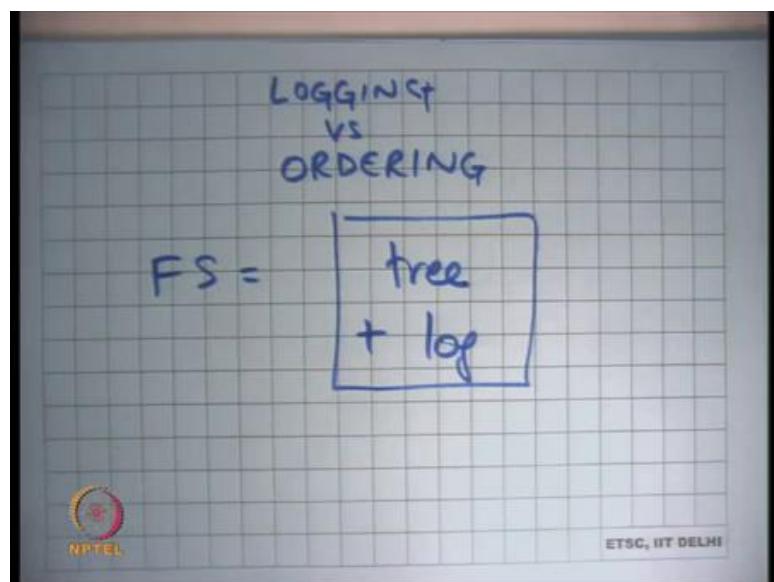
So, even though still you are having 2x writes you are writing each block to the disk twice, the overhead is not 2x right because the first write is part of a large sequential writes. So, it is almost free and the second write is also not as expensive because you have

a lot of scheduling potential because you are doing lots of in flight IOs for application of to the FS tree.

So, the number of seeks or latencies a that you pay is still roughly 1x right even though you are writing the blocks twice to the disk the amount of overhead or the number of assuming that you are counting over ahead as number of seeks you making to the disk is still 1x alright.

So, that is you getting; so, basically with logging you have used the characteristics of the magnetic disk to ensure fast cache recovery yet be able to maintain the same level of performance. The advantage of logging over the previous methods that we saw which was ordering.

(Refer Slide Time: 05:46)



So, if I just compare logging versus ordering as we saw that the performance difference is not much, logging is almost the same performance as ordering. In fact, its little simpler you do not have to worry about; so, you know recall that ordering had this problem that some invariants will get you know some inconsistencies can arise.

For example, free space they can be memory disk (Refer Time: 06:11) leaks or there can be in the case of moving a file from one directory to another it is possible that the same file is pointed to by two directly. So, ordering hard possibilities of inconsistencies,

logging does not have them, logging ensures the atomicity of the entire operation across crashes.

Also, the other biggest advantages basically recovery at recovery time you do not need to do a full global files system scan right. Recall in the ordering case to check my invariants I had to actually troubles the full file system in a global way and that was becoming very expensive. And, we said it can take an hour for the sizes of disk that we have today which is 100 of gigabytes to you know 1 or 2 terabytes, its it can take up to hours to do this and that is becoming very impractical as we go along.

But logging has does not have this problem right, you do not have to do a global files system scan right. What do you need to do to recover from a crash? You need to scan the last few transactions in the log right and see which of them have not yet been applied to the disk right.

So, if we are going to look at how exactly the crash recovery works, but you can already see that you do not need a global operations; all you need to look at is the last few blocks of the log, yes question.

Student: Sir, does log exist in the file system?

Does a log exist in the file system? Yes, I mean log is part log is the structure on.

Student: (Refer Time: 07:34) file.

You can consider the log as a separate file in file system, but I mean its it has special semantics. It is not your you know it does not have the same semantic. So, for example, the user cannot see the log right, the user cannot read or write directly to the log, the log is basically being written by the system by the kernel.

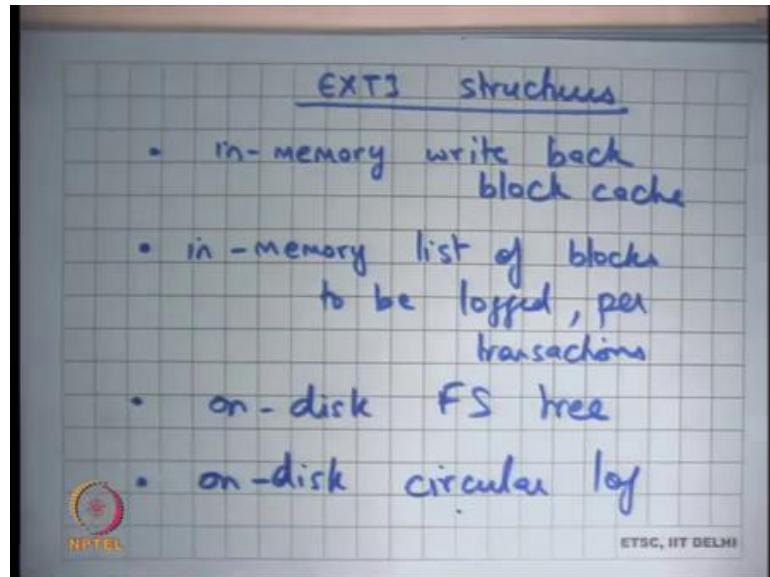
Student: Sir, I am asking you suppose something gets corrupted while writing to the files systems structure. So, can we still access the log to recover from that?

So, depends on; so, the question is if something gets corrupted while you are writing the file system structure can we recover it by using the log? So, I mean in generally yes, but I mean depends on what kind of corruption you are talking about right. I mean the whole

idea of the log is that you; so, basically your file system now is equal to a the tree which is you know tree of inodes and blocks and directories and so on plus the log right.

So, your tree can become inconsistent, but the total of tree plus log will never be inconsistent right. So, that is the guarantee that you know if you look at this whole structure as one whole thing tree and log then you know the invariance across these things are always maintained. The tree itself can get inconsistent and that is why the you know when you, if there is a crash then the log can basically correct the tree (Refer Time: 09:02). So, let us look at the ext3 structures.

(Refer Slide Time: 09:13)



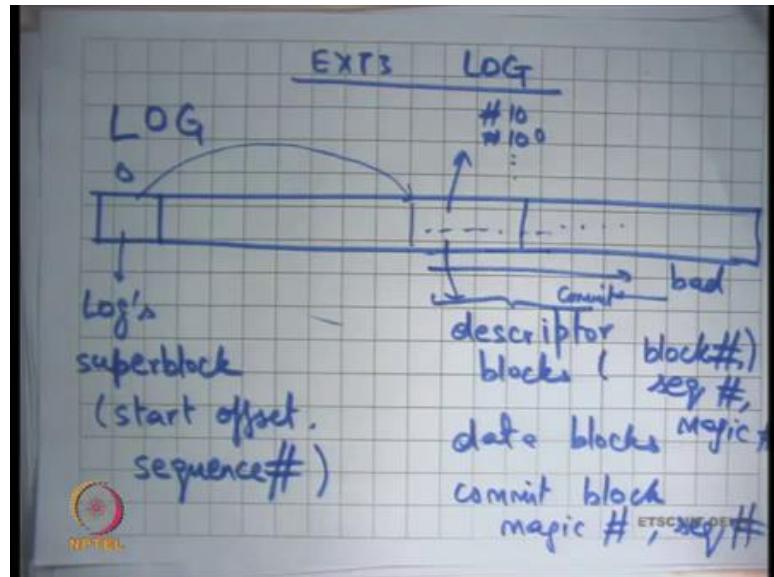
So, there is an in memory write back buffer cache or block cache right, we already know this. We have seen this was present even without logging, you basically have a buffer cache which we need and needs to be right back for performance. You have now an in-memory list of blocks to be logged per transaction.

So, as your as the disk operations are happening you make you are keeping an in memory record of these are the list of blocks that belong to this that have been dirtied, that have been modified and they belong to this particular transaction. So, when this transaction commits these are the blocks that need to be flushed to the disk right.

So, you need to maintain this data structure in memory alright and then of course, you have the on-disk FS tree and then you have the on-disk log. And, this log is maintained

as a circular buffer because you know the log cannot keep growing in you know infinitely. So, you basically just wrap around the log and you also keep freeing the log as you apply the log changes to the file system tree, we are going to see how.

(Refer Slide Time: 10:45)



So, let us look at the ext3 log. So, ext3 log let us say I if I draw the ext 3 log this is my this is not my disk, this is my log on the disk. So, this is the part of the disc which stores the log let us say alright and the log has let us say those 0th block in the log is the logs superblock. This superblock is different from the file system super block, this is the log superblock.

By superblock I basically mean basically mean that it contains meta level information about the log right for example, where does it start right. So, this will contain you know start offset and a sequence number. The sequence number indicates what the current, what the sequence number of the current of the transaction that the first transaction at this start off set right.

So, the idea is that you have you maintain global sequence number and each time you close a transaction you increment the sequence number for the next transaction. So, one after another the sequence each transaction will have a new sequence number right. So, this is basically pointing somewhere here the start of set and you can find the log starting here ok. At any time, you reach the end of this area you wrap around like this alright.

What does the log have? The log has two types of blocks descriptor blocks and data blocks. So, descriptor block basically says that the next block; so, it just describes the next block for example, or in the next few blocks.

Basically, says the next few data blocks are for the next few data blocks are for block numbers x y z and so, these are the data blocks. So, for example, you can say here is a descriptor block which says number 10, number 100 and so on. And, then after that you have the blocks the respective blocks, the contents of the respective blocks right.

So, there are two types of blocks, there is descriptor block and there is a data block. The data block contains a full data, in the descriptor blocks describes what data it contains. Are the descriptor block similar to inode in structure? No, not at all, descriptive blocks are just something specific to the log which just says that no. So, basically the log contains all the dirty blocks right that need to be applied to the file system.

So, the descriptor block just says that these are the dirt you know the next few blocks are data blocks and these are the these data blocks are the contents of block numbers xyz. So, you know exactly where to apply these blocks that is all. So, the descriptor blocks contain information like block number alright.

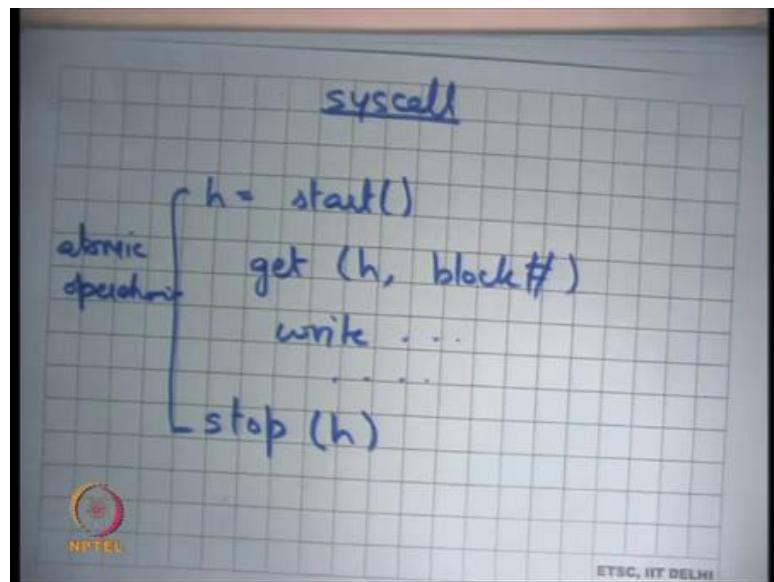
So, which block it is we have already discussed this, it also contains information about sequence number we going to see why need it. So, it every descriptor block contains a number which says which transaction do I belong to right ok. So, this sequence number is the same. So, as the sequence number in the block and we going to see why it is needed, why do we need a copy of the sequence number in the descriptive block. And finally, we have something called a magic number.

By magic number it basically means that there is some number there which says that yes, it is a descriptor block it is some identifier. So, it is not some random block, it is it if the magic number is present you can be sure that it is your log block, it does not have any garbage in it. So, it is just an identifier which says it is a block of ext3 log.

And so, in general the magic number is used in many different places and the purpose of a magic number is to identify or to disambiguate a valid block or valid piece of data from completely garbage data ok.

And then so, that is there is descriptor block, there is data block and then there is a commit block alright. It basically says that you know at this point this is where the whether the transaction committed or not right. And, the commit block contains once again it contains a magic number and it contains the sequence number of the transaction, that it commits .

(Refer Slide Time: 15:21)



So, just to review how does the syscall work, it basically says let us say I wanted to make a disk write. So, I will say h is equal to start, this is like start transaction as we have discussed last time or start operation as we have discussed last time. Then you are going to say you know get h block number to indicate that you are dirtying this block you are going to you know write to these blocks and then you going to say stop h right.

So, this is how you basically say that this operation should be made part of the current transaction. And, this operation should be atomic with respect to crashes right. So, it is basically its so, all you have to do is basically look at areas where are you doing multiple disk writes and identify points where you want to say that these multiple disk writes should be atomic with respect to crash failures and then you put a start and a stop around them right. You know one simple thing to do could be that you put a start and a stop around at the big at the start of the system called and at the end of the system call always, you know that is one safe option to do also right.

So, this has a lot of similarity with transactions as we have discussed earlier right, you say begin transaction and end transaction and it just runtime system that takes care of what are the things that you modified and whether they going conflicts or not etcetera similar thing here right. Because, their transactions are not logs you do not have to worry about fine grained versus coarse grained, you do not have to worry about deadlock something like right. So, you are get all the advantages of transactions alright; alright so let us see.

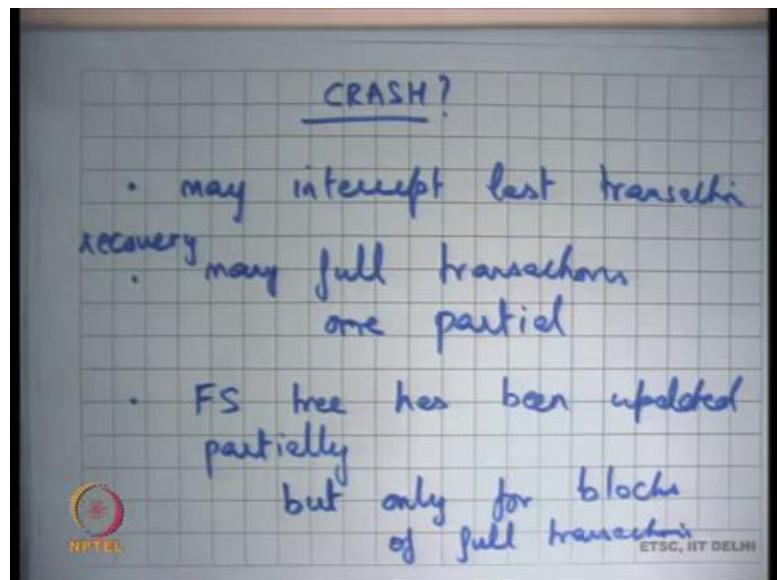
Student: (Refer Time: 17:07) reference. So, here also like multiple transactions can execute at the same time and then it later checks whether there is some discrepancy and reverts back.

So, the question is you know in these transactions it's possible that multiple transaction is happening at the same time and if there is a discrepancy then you rollback. Notice that these things here are not what I am calling transactions, these are I am these are the terminology I am using for these are atomic operations.

And, their multiple atomic operation that in go on concurrently and they are all part of a single transaction right, atomic operations do not have any conflicts with each other. So, we are not talking about you know. So, one atomic operation and the another atomic operation as long as they touching different buffers have absolutely nothing to do with each other. But we want to make sure that you know their atomic with respect to crash failure. So, here atomic is not with respect to each other, here atomic is with respect to power failures right.

So, and so, you can you know so, this the way you deal with that is different. So, now, you put lots of you allow these atomic operations to execute concurrently, but you put them all in a single transaction. And, that transaction is made atomic with respect to power failures, that is what you are doing alright.

(Refer Slide Time: 18:31)



So, let us see how what happens on a crash ok. So, it may interrupt last transaction while it was writing let say the transaction has decided to commit, its writing the transaction to the log and before it could write the commit log, the power went off that is possible alright. So, it is possible that you have many full transactions, when you recover; at recovery time you see many full transactions and one partial transaction in the log right ok. Also, you may see that the FS tree has been updated partially alright.

So, when you recover from a crash these are the possible things that can happen, you can you know you can see lots of full transactions and you can see one partial transaction. And, you can see that the FS tree has been partially updated which makes it inconsistent, but you know together with the log you can you know that tier structure is consistent.

So, but the what is the other thing? Can it be updated partially? Can it be updated with the transactions that have not to yet completed? So, the FS will up only, but we update a partially, but only for blocks belonging to full transactions. So, the then we again following is that after only after we commit a transaction to the disk, do we start applying that transactions blocks to the tree?

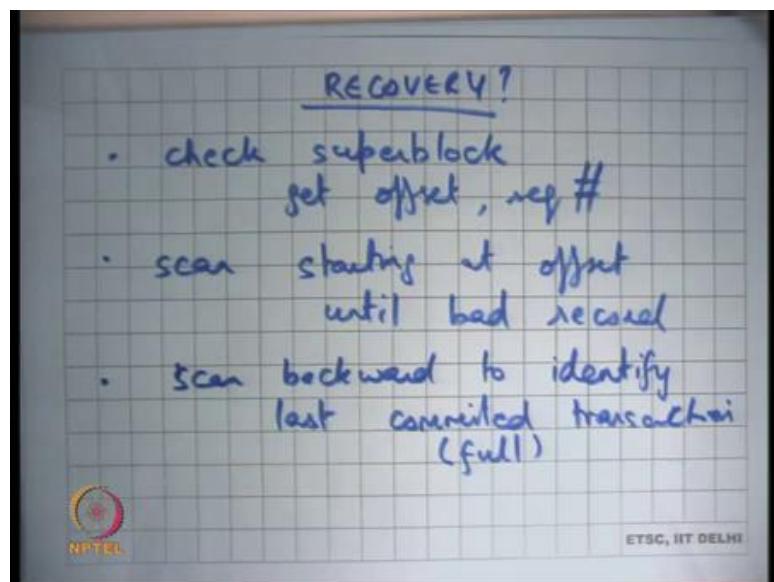
So, the one that is partial none of it is applied to the filesystem, you can be sure of that; its only the ones that I have completed whose blocks have been applied to the file system. And so, if partial blocks have been applied to the filesystem no problem you can you know because application of is completely idempotent.

So, you can reapply them, this is the same contents that you are writing again. So, all you need to do is look at all the full transactions and reapply all of them to the filesystem tree. And, completely ignore the partial transaction, it's as of the partial transaction did not happen at all right.

So, what kind of semantics is giving the programmer? It basically saying that if you for example, create a file and there is a crash after you know even though the create system call return. And, after that you have displayed something to the user saying you know I have created the file etcetera and there is a crash and then you come back again there is no guarantee that the file is actually on the disk.

Because, the transaction which contains this create operation may not yet have committed. And, let us say its committing every 30 seconds then that is not committed. But, given that you have some reasonable interval for the commit transaction, then be show that something that you wrote you know few minutes back or 1 hour back is on disk alright.

(Refer Slide Time: 21:51)



So, let us see how does recovery work. So, first check superblock ok, to get offset and sequence number right. Then scan starting a offset until back record right. So, basically at recovery time I look at the super block to get the start of the transaction and then I just keep scanning forward till I find something that is bad. What do I mean by bad? No. So,

by bad I mean I see a block that does not have the same sequence number that I am expecting ok.

So, how do I know that something has not committed? Basically, I am going through all the blocks and I am looking at the sequence number of these blocks right and then I suddenly see a sequence number that is older right. So, that basically means that this transaction could not write all the blocks right. So, the sequence number is disambiguating between blocks of the current transaction and blocks that may have been written earlier recall that it is a circular log right.

So, if you know there could be some rights that I have living from the previous log. So, you just scan the log till you till you say something wrong and something wrong could mean a bad sequence number or it could mean a bad magic number. So, it is possible that you know the disk had completely garbage, the log had completely garbage contents in the beginning and then you basically you know started writing.

And now the sequence number is see somehow correct, but the magic number you know there assumption is the magic number cannot be correct, if you know if it is some garbage. The probability that the magic number is also correct is very small basically ok. So, you basically start here, and you keep scanning till you find the first record that is bad and then you scan backward from there to find the first commit record right.

So, you just go like this to find something which is bad and then you find the first commit record after the before this and these are the transactions that you want to apply to the file system tree. And, when you apply this transactions to the filesystem tree you can free these blocks. What does freeing the blocks means?

Student: Update.

Update the start offset right. So, you basically when as you apply the transaction you also update the start offset to point to the next transaction and so on ok. So, that is basically how recovery works. So, notice that once again if there is a crash you identify a bad block by either a bad sequence number or a bad magic number and that indicates an incomplete transaction. And, from there you go back to figure out what are the last full transaction that you saw and that is the one you apply to the filesystem.

And, once you applied to the filesystem tree you although free up the space by advancing the start offset and this is a circular log so, you just wrap around and so you basically keep on using the log. It is possible that you know while you while the transaction was executing the log is completely full right.

So, let us say you are executing the transaction on the log, you are writing disk blocks on the log for this particular transaction; let us say your system call that involved deleting many files. So, lots of different writes are happening and so, you basically have a large transaction and this transaction is not being is not fitting in the log. So, what do you do?

Well. Firstly, if the size of the transaction itself is larger than the size of the entire log space that you reserved for this purpose then there is nothing you can do. So, you should and the programmer should ensure that the size of a transaction can never become larger than the size of the log; either the log should be allocated large enough about you know the size of the atomic operations that you are taking it should be limited, it should be bounded to some size number 1. Number 2, but still it is possible that while you are doing something; so, the you know you hit the end of the end of the log.

And, clearly you know you hit the end of the log if you ever you know reads the start offset again, you will wrap around, and you read the start offset again. So, it is very easy in memory you can figure out that you at the end of the log. So, what you can do is you can just commit the previous free the previous transactions do not commit, but they already committed.

But you free the previously committed transactions, by freeing the previous commit transaction means you apply their contents to the file system tree, and you have advance start offset. So, that the this is more space that is freed up in the log.

Why do we have the start offset? To indicate a start of the log. Why is not the log start log starting at log number 1? Because, it is a circular log right. So, I want that you know; so, there is a producer to the log and there is a consumer from the log and so, it is a circular buffer. So, you need to (Refer Time: 26:39) both head and tail right. So, let us say you just had a log with always started from 1 right. So, now if I wanted to a free a transaction, it would mean copying the just for the log all the way. So, it would have been a global copy to block number 1 to free a log.

So, let us say I wanted to free the first transaction in the log right, then I have to copy all the transaction starting from transaction number 2 that is a very expensive operations. So, the better way to do it is maintained as a circular buffer and have a head pointer which is the start offset and a tail pointer which is the which is not tail pointer here recall right.

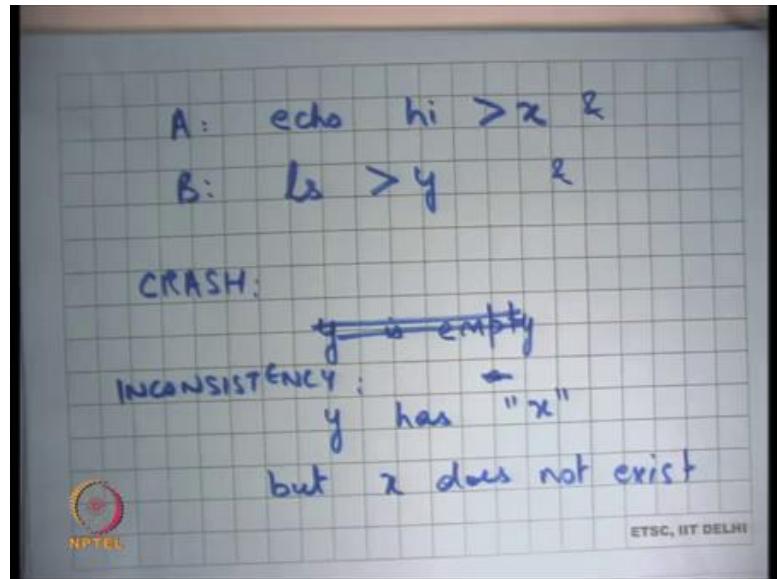
You are using you are finding you are inferring the tail pointer using the sequence number and the magic number ok. So, a scans starting at offset until bad record, scan backward to identify last committed transaction or you know the this is the full transaction, this must be a full transaction.

And, then you can apply those transactions and completely discard last transaction ok. Also, even without recovery the there should be an asynchronous thread that is bring up space from the transaction log right. The asynchronous thread works similarly, it starts at the start offset and looks at the first transaction.

You know; so, how does the asynchronous thread work? It just looks at the start offset and looks for the first transaction and freeze it up, by freeze it up it basically apply that is transaction to the file system. If it finds that transaction is really small, it can club multiple transactions and free all of them together right to have better to this scheduling; so, this is the kind of freedom it has to get better performance.

So, as I have discussed so far, we are logging the entire file system including the inodes, the free block lists, the bitmaps the and the data logs themselves alright. Let us look at you know what kind of semantics does it give us.

(Refer Slide Time: 28:51)



So, let us say there were two operations: echo hi > x and then there is ls > y right. If you execute two commands one after another and let us say they execute concurrently. So, you know let us say I put an ampersand operator here. So, they can execute concurrently, there is not sequentially here. And so, basically if there is a crash, the consistent so, I should either see that y is let us say initially the directory the current working directory is not there. So, either I should see y is empty and right.

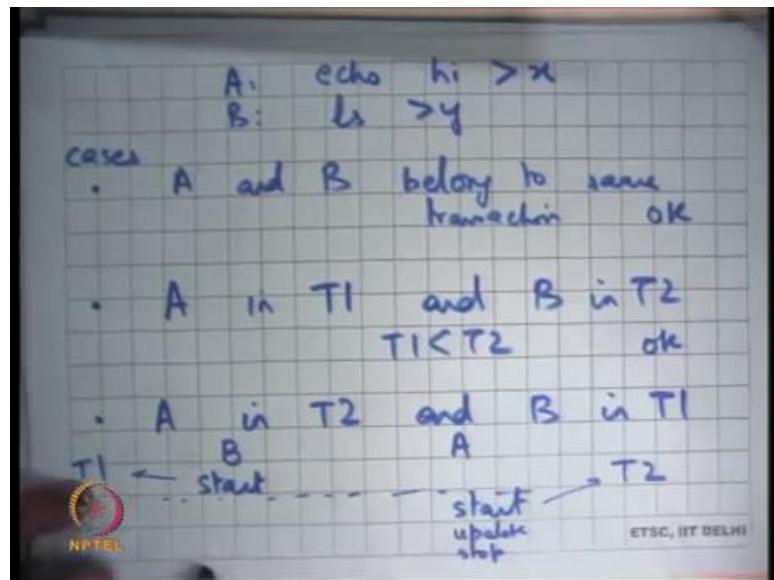
So, what I should not see let us see is that y has x written into it, but so, they could an inconsistency could be that y has x, but x does not exist. So, this would be an inconsistency, I created a file in the directory and then I did a ls to read the contents of the directory and wrote it to another file y right.

And, you know if there was no crash then definitely you will always see consistent behavior because you know all your answers, all your answers that you getting whether what exists in the file system and not being surf in the buffer crash.

But, if there is a crash then you know it you should not see this kind of a behavior that y seems to have seen the file x, but the file x does not actually exist right. So, there should be some civilization with basically says either the file x does not exist and y does not contain x or the file x exits and y contains x, but it should not be that the y contains x for the file x does not exist right. So, that would have been an inconsistency.

Now, let us see whether this can happen in the ext3 transactional system that we have discussed so far ok. So, let us say these two operations are A and B and they are happening concurrently. So, and we basically want to make sure that this kind of an inconsistency does not happen. So, we do not like this.

(Refer Slide Time: 31:20)



So, let us say so, let me just write it here A echo hi greater than x, B ls greater than y. So, let us take cases if A and B belong to same transaction. So, my question is if A and B belongs to the same transaction can this inconsistency happen on a crash ok.

So, I see some people nodding the heads; so, it cannot happen. Why cannot it happen? Because, if they belongs to the same transaction either the transaction would have committed in which case you would see hi in x and x in y or the transaction will not have been committed in which case you will not see either.

So, you will never see this inconsistency right either the transaction would commit in which case you will see both the updates and the transaction will not commit in which case you will not see any of the updates. It will not happen that one of the update is seen and the other is not seen. So, if they belong to the same transaction there is no problem, let us say case 2: A in T1 and B in T2, where T1 commits before T2 alright.

So, A is in T1 and B is in T2. So, in this case you write so, if the crash happens it is possible that T1 has committed, but T2 has not committed. In which case what you will

see is that the file x has been created, but the file y has not been created at alright that is also fine right. So, there is no inconsistency of this type that y has been created and it seems to have x, but x does not exist; so, that is also not a problem at alright.

So, basically if A is an a transaction T1 and B is an transaction T2 there are few things that been happened either at the crash time none of them have committed in which case none of these updates had happened or it's possible that both of them have committed in which case both are the updates had happened.

Or, it is possible that T1 has committed and T2 has not committed in which case As updated has happened, but Bs update has not happened in which case you know the y has not been even written to the file system right. So, there is no y, but x has been created, but y has not been created. So, that is not an inconsistency, the inconsistency y has been created with the contents which are inconsistent.

So, even this is right. So, let me just say and now let us say is it possible that A is in T2 and B is in T1 right. So, A is in T2 and B is in T1, can this inconsistency happen in this case?

Student: Yes.

So, there is an answer yes. Why?

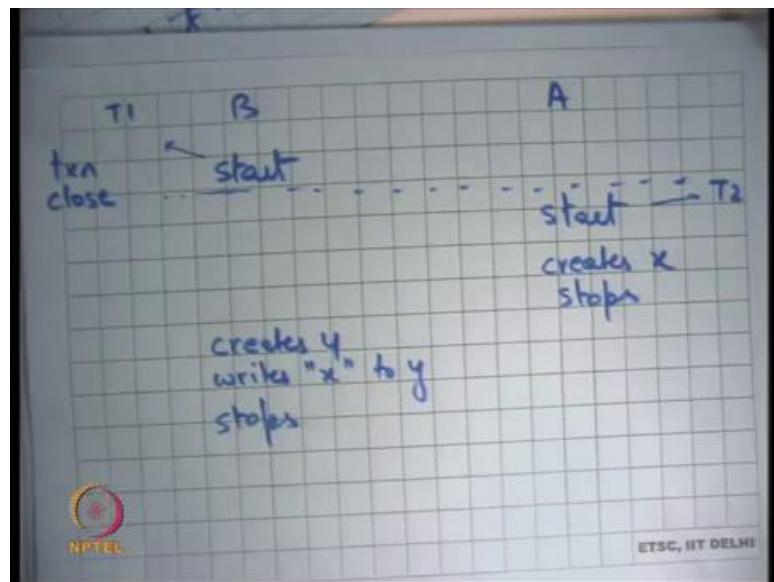
Student: Because, since we are first writing to y; so, basically not of so where the x was not there.

Right. So, it is possible that y is in; so, because it is for now if there is a crash and so, once again if there is a crash then either none of them have committed in which case we are or both of them have committed in which case we are ok. But, if T1 has committed, but T2 has not committed then y's contents have been written to disk, but x's contents have not been written to disk alright.

So, but you may say you know if y's content has been written to disk, but y must, but y must have seen x's update right. So, how can this happen? It can happen if let us say; let us say this was B and this was A. So, B started first alright and then A started later. So, this and here was the closed point; so, the transaction got closed after this.

So, this B's transaction gets into T1 and A's transactions gets into T2 right. Because, there was a transaction closer between these two starts, recall that when you close the transaction then all the existing transactions all the already opened operations are completed and all the future operations are made part of the next transaction right. So, if B's start happened before A start then B could be in T1 and A could be in T2. Now, A could have you know made its update and then stopped and then B will let me write it on a different page.

(Refer Slide Time: 35:59)



So, let us say this is B A and this is start and this is start, this a dotted line which says close transaction close right. And so, this is part of T1, and this is part of T2 and then this updates creates y and then stops alright and then this says creates y writes x to y and stops right. So, even though B could see A's update, B happens to be in a transaction earlier than A right. So, in this case B could see A's update, but because B started before A, B is in a previous transaction.

So now, it is possible that you know at this point I commit T1 and you will see B you will see the file y containing the contents x, but the x, but because T2 has not committed you do not see x in the filesystem. So, you see an inconsistency and so it's possible that this happens. So, the question is in this case we are running two transaction simultaneously.

So, recall so, let us look at what are the transaction mean in the ext3 sense. A transaction means that you know only one transaction is open at anytime. All operations that have started during at this point get assigned to the current transaction right. Then you close the transaction at some point you would side have close the transaction. So, all operations that start after this will start in the new transaction.

Student: But we do not close until all the operations and then transaction are closed also.

No. So, by close I do not mean commit. So, there is a difference between transaction closure and transaction commit. So, I must decide some point where I commit the transaction right, close the transaction.

Student: But sir I am saying (Refer Time: 38:02).

Right. So, let us listen understand this. So, let us I had I close a transaction and so, there are some there is possible at some ongoing operations. So, what you are suggesting is that when you close a transaction you do not take any new operations and you wait for all the existing transaction operations to finish.

Student: So, sir I am saying like suppose in operation has bigger in current transaction like in this case we have taken start from D.

Ok.

Student: So, we will not close that transaction until we get stopped for that.

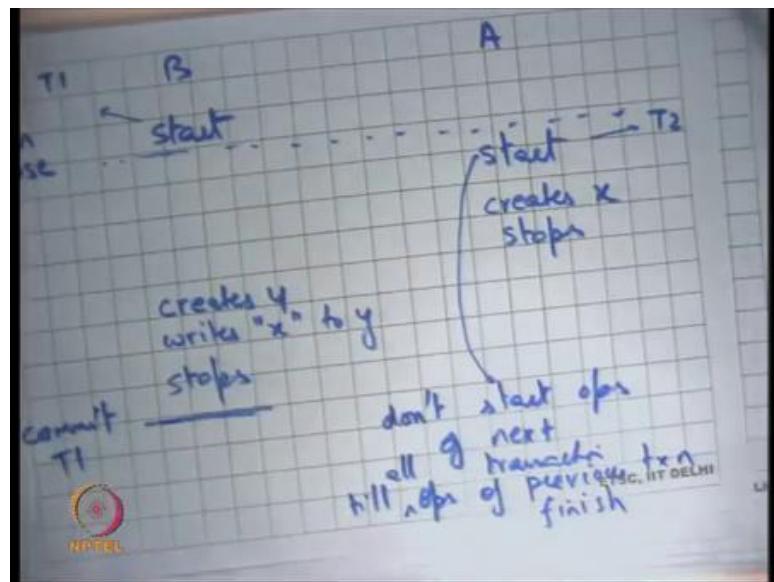
But let us say then A starts you know let us say a wait for the transaction to close till here and then you know this transaction start. So, this transaction becomes part of this existing transaction.

Student: Yes, Sir.

But this can continue forever right. So, let us say there are lots of disk operation happening. So, I will never get a chance to close the transaction right. So, the idea was that I just decide that I have close the transaction and then all the future operations become part of the next transaction. If I wait for all ongoing transactions to finish before I close the transaction then it is possible that I never get a chance to close the transaction, because there are lots of ongoing transaction that are going on right.

So, the idea is that I just decide to close the transaction which means that all the transactions that are already started they will get through in this transaction. But all everybody who has not been able to catch the bus will get the next bus basically right. So, you know that way you basically are sure that you can close the transaction at periodic interval, otherwise you will never be able to do this.

(Refer Slide Time: 39:40)



So, to avoid this inconsistency you basically add one more rule to this that do not start ops of next transaction till ops of previous transaction, till all ops of previous transaction finish ok. So, the idea is that you do not till this one finishes; so, it similar to what you are saying accept that you still have guarantees on you know you can choose when you want to close, you do not have to wait for things to be able to close a transaction.

So, you say that if you have decided to close this transaction before you start exuding any operation at the next transaction you are going to wait for all the operations that are ongoing in the previous transaction to finish, to avoid to prevent these inconsistencies.

So, in this case you will make sure that this start happens after this stop right, in which case you have complete you do not have any inconsistency, you have serializability in the behavior with respect to power crashes right. So, the idea is you do not start the operations of the next transaction till the operation that the previous transaction has finished.

Notice, that I am just waiting for the operation operations of the previous transaction to finish, I am not waiting for any disk writes right. These operations may happen in memory right, I am just waiting for them to finish. So, that what I read from the file system including the buffer cache is serial with respect to the previous transaction ok. So, it's not a very long wait, you are not waiting for the transaction to commit, you are not waiting for any disk write, you are only waiting for the operation that the previous transaction to stop, yes question.

Student: Sir, we have done this actually prevents the kind of inconsistency that we are talking about and there, but sir in this case what happened is that x did not, could not get written into (Refer Time: 41:41) because, there was not any x.

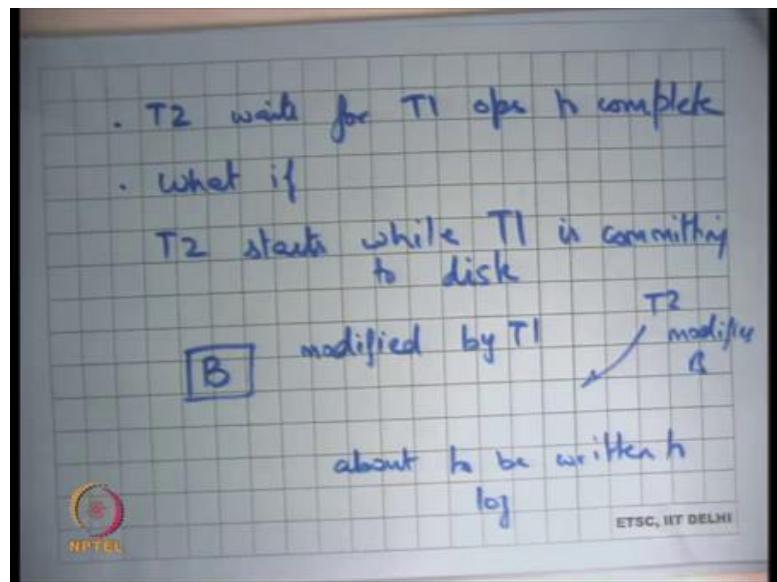
So, in this case what happens is that x did not get written into y and if there is a failure. So, it's so, basically this inconsistency you know avoided, but you are saying that in this case x has not been written to y.

So, is not it that inconsistent, well that is not inconsistent; what is consistency mean there is some serial? So, there is some serial order in which things appear to have happened right. If you delay this, the serial order seems to happen the seems to that seems happen is that B happened before A right.

It is completely acceptable if B happened before A, it's also completely acceptable if A happened before B, but what is not acceptable is that after A crash it seems like A has happened partially, but B and B has happened partially right. So, that was a inconsistency. So, A seem to have happened partially you know the contents of the directory seem to have changed when as I had read them, but actually that I directory does not seem to have that contents; so, that is an inconsistency.

But B happening before A and A happening before B both are legal. So, those that is consistent and so, what is this doing is basically making it look like B happened before A and that is totally fine.

(Refer Slide Time: 42:55)

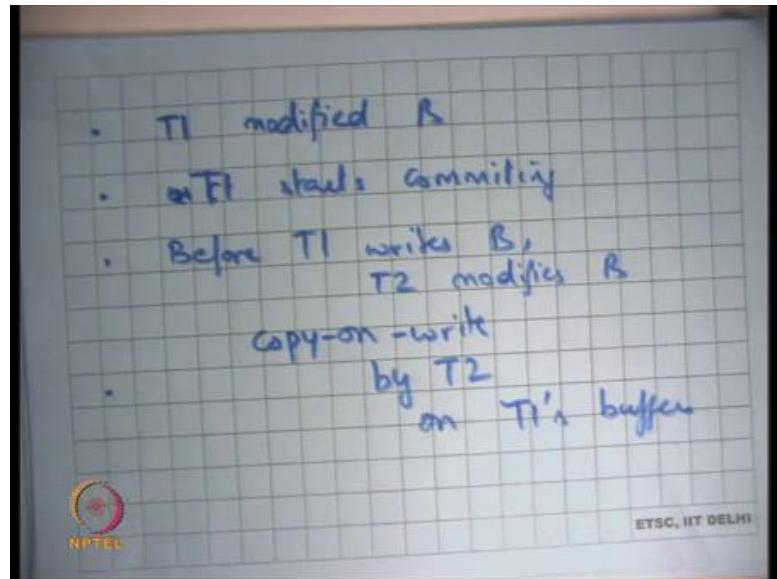


So, basically, we said that T2 waits for T1 ops to complete right, but it does not wait for T1 to commit, T1 can you know commit later that is not a problem. So, what happens if T2 starts, what if T2 starts while T1 is committing alright. So, T1 is committing to the disk and T2 has started its operations ok. So, what has what is some bad thing that can happen?

Well, in this case what can happen is you know you have decided that you are going to write this buffer to the filesystem tree and that that buffer basically at to the log actually. So, let us say there is some buffer B that is modified by T1 and about to be written to log.

But before it is return to the log T2 modifies B right. So, if you are allowing T2 to start running before T1 has committed then it is possible that a buffer that belong to that was dirtied by T1 is again modified by T2. And so, what can happen is that if you if T2 modifies B here then what gets written to log is are the contents of T2 which include T2's modifications right. Let me just rephrase it.

(Refer Slide Time: 44:47)



So, let us say T1 modified B writing T1 starts commit, starts commit means starts writing its modified buffers to the log right. So, before T1 writes B T2 modifies B. So, what can happen? The value the contents are get recorded on the log contain T2's modification and that is not something that you want right. So, what is a simple solution? So, you just make a copy of this buffer B right.

So, you basically say that this buffer B belongs to the commit buffers of a previous transaction and these buffers has not yet been written to the log. And, if there is a if so, you do copy on write sum basically by T2 on T1's buffers ok. So, if T2 tries to write to a buffer that belongs to T1 and has not yet been written to disk you make a copy of it right.

And, it is a most recent copy that is the that is the valid copy, the previous copy you know it's only for T1 to commit to disk yes question.

Student: Sir, why cannot we use the logs, you are already having the logs or buffer?

Why cannot we use the logs? That is an interesting question; you want to keep a log on all these buffers while you are committing this transaction T1.

Student: While, the transaction is while I am writing by buffers at this program, I will release the logs only at the transaction gets committed.

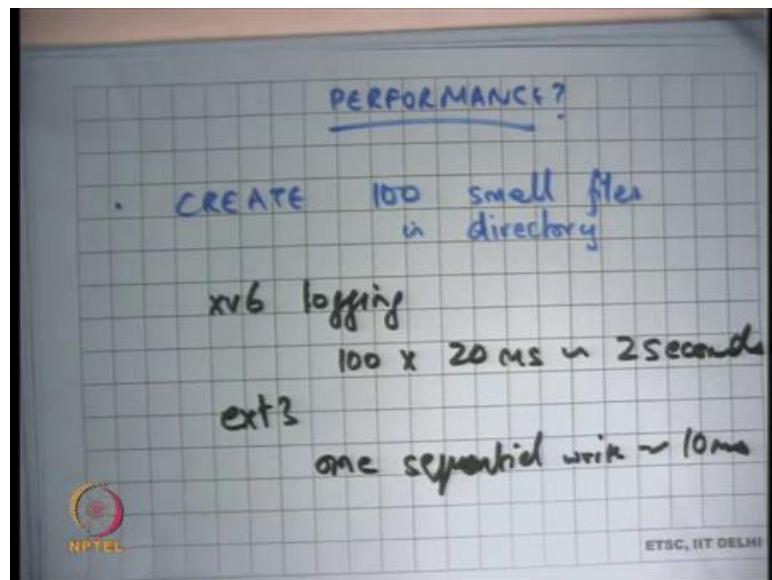
So, here is the suggestion I will release the logs only when the transaction gets committed, is that an option? Well, that is an option that seems like an option to me, you know you basically making sure that till the transaction commits you basically log all the buffers that belong to the transaction. But, is not it that too long to hold onto the log of a buffer, right?

You basically want that thing should happen concurrently as much as possible, imagine that you know you are constantly writing or modifying the same file or you are constantly adding or removing files from the same directory..

Then you know if that directly happened belongs to the previous transaction you are holding logs to it and next transaction just gets completely serialized with respect to the previous transactions. So, that is not a good idea, the better idea is to basically you know do this versioning of the blocks. So, you are basically doing versioning you are saying this block has you know has the T1's version of it is this and this is the current version of this block.

And so, that is some much more efficient way of doing things. So, its once again I think it is a classic trade off example of a trade off between transactions and logs right. Here transactions are provide more concurrency and yet come at a small cost right because you are because here transactions involved disk accesses, so doing this figuring out of you know what is needs to be copied etcetera is possible alright.

(Refer Slide Time: 48:00)



So, let us quickly look at the performance of ext3. So, let us say I create 100 small files in a directory. In the simple logging or I am going to call at the xv6 way of logging, xv6 logging it would have involved each create would have involved 5 or 6 or 8 disk writes and a commit. So, each write would have been 1 commit and that would have been let say 100 into 5 or 6 writes.

So, let us say you know 20 milliseconds is roughly 100 into 20 milli seconds is 2 seconds right. Is that right? And, but with ext3 all these 100 small files creation happens on in inside memory and then there is one sequential write that is 100 10 milliseconds.

So, 2 seconds versus 10 milliseconds and then there is an asynchronous copy from the log, or you know asynchronous application to the filesystem tree, which is not in the critical path and so, that is can be done later. So, it much faster in that sense. Also, you know I say one sequential write, but you may want to do two sequential writes.

One sequential write for writing all the blocks in one go and then after they have been written then you write the commit record after that. You do not want the disk to also schedule the commit record right, you do not want the disk to reorder the commit record with respect to the other data blocks.

If it does that then you have a very bad situation, the commit block has been written, but the data blocks have not been written. So, you first give the disk all the data blocks to be written and the descriptive blocks; after that have been written the disk say I have written them then you give him the commit block. And so, there may be two revolutions there ok.

Let us continue this discussion or finish this discussion next time and I am also going to discuss security and access control next time.

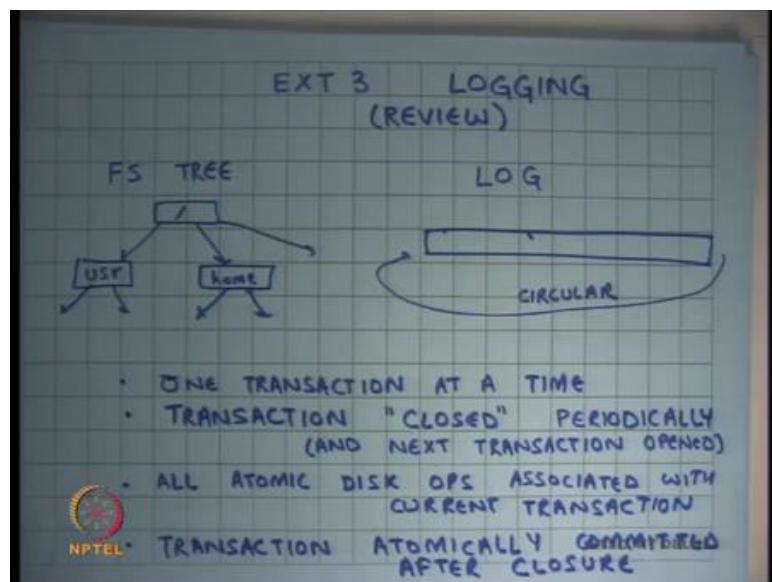
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 36
Protection and Security

Welcome to Operating Systems, lecture 36, right. So, so far we had been looking at logging as a way of maintaining, ensuring that the system can recover after crash, in other words ensuring that the disk does not get so corrupted that you cannot even recover your data out of it, right.

So, we have the first thing we had looked at was ordering where we said that we are going to order the disk operations, as they go to disk in a certain way, so that we do not lose data. We may have space leaks, temporary space leaks, but they can be fixed whenever we. But there were problems with that the problem was that recovery was really slow and the other problem was inconsistency was still possible.

(Refer Slide Time: 01:01)



So, logging was other way where basically we the file system consisted of a tree which is your regular file system tree and a circular log. You will have one transaction at a time. So, I am talking about the ext3 file system. You will have one transaction open at a time which you will close periodically. So, let us say you close the transaction every 5 seconds for example. And all the atomic disk operations that are started at any time will

be associated; will be tagged with the current transaction with the current open transactions.

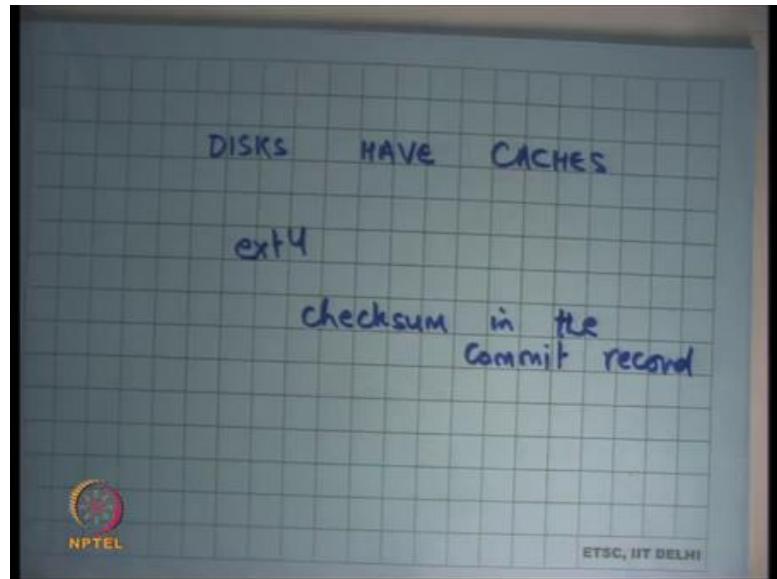
The atomic operation will either belong completely to one transaction or not belong to the transaction at all. It is not possible for an atomic operation to be half in one transaction and half in another transaction. Once you close the transaction you also open the next transaction. So, all atomic operations that start after you close the transaction will belong to the next transaction. All atomic operations that started before you close the transaction irrespective of whether they have completed or not will belong to the previous transaction. If they have not completed the transaction will wait for those operations to finish before it starts committing them to disk; alright.

So, all atomic disk operations associated with the current transaction and transaction is atomically committed after closure. And as we saw last time commit, committing the transaction, before you start the commit of a transaction you have to wait for all the atomic operations to finish number 1. Number 2, you also want to make sure that no atomic operations of the second transaction or the next transaction has started before all the operations of the previous transactions are finished, otherwise inconsistencies can result and we had seen some inconsistency last time that can happen, ok.

And we also said you know even though, so in this case we log old data blocks atomic commit of the transaction is implemented by the right of a special commit record at the end of the log and assuming that this right of the commit record is atomic you are basically sure that either that transaction will be atomic atomically committed or it will not be committed at all; in which if it is not committed means nothing happened if it committed it means everything happened. So, there is no intermediate state. So, there cannot be any inconsistency assuming that commit is atomic.

So, we had been so far assuming that the commit has atomic; we have been so far assuming that the disk behaves exactly like the way we expected to be here. So, for example, we assume that if we tell the disk to write to a particular sector it has written to it before it actually comes back to us. But the truth in fact, is that the disks have caches, right.

(Refer Slide Time: 03:43)



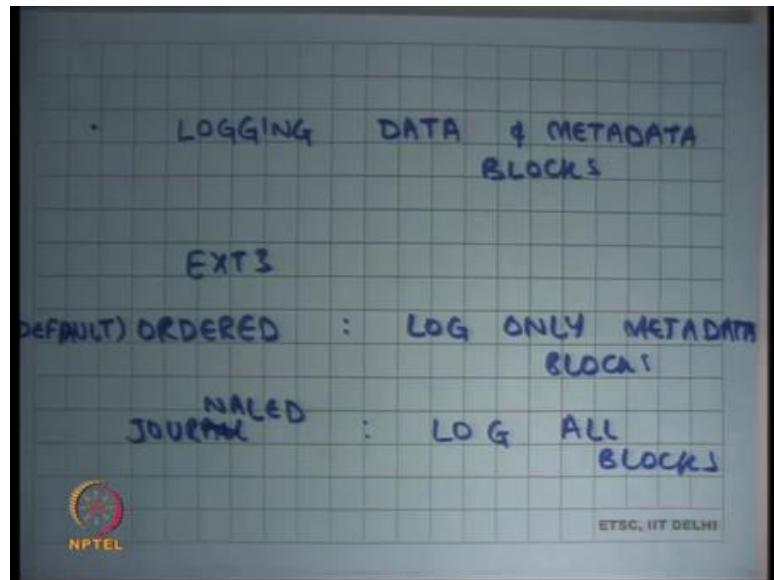
So, in the real-world disks are not really truthful, right not necessarily truthful. You know you can imagine that this manufactured by different devices different manufacturers and for better performance these manufacturers are putting special as you know volatile caches which means you know they are either using technologies, the same technology that we are using for DRAM for your main memory into your disk. And so, even though the disk has said that it has finished it may be doing some optimization at the back end where it may not have written to the magnetic platter, right to the actual magnetic platter.

So, if something like that happens then you know all your reasoning about correctness actually goes for a toss, right. Because if you know if you ask the disk write the log and wait for the disk to respond and then you ask the disk to write the commit record. And, the disk instead internally has cached the first write has not actually committed to the magnetic platter, and it has it commits the commit record before the actual contents of the log then when you recover you will see that there is a committed transaction, but the transactions contents may actually be junk, alright.

So, these kind of things are possible and so, ext3 does not deal with these kind of problems, but the next file system ext4 introduces checksums in the commit record, right. So, the idea is that the record, the commit record will have a checksum or some kind of a one-way hash of the contents of that transaction, right. So, when you are

recovering you can check that the checksum actually matches the contents of the transaction and if not then you can say that you know something fishy has happened and you should not be actually doing that transaction, ok. So, that is one way of dealing with these kind of issues, alright, ok. So, as we have discussed logging, we were logging whole data blocks both data and metadata blocks, right.

(Refer Slide Time: 06:05)



So, if I were to look at the file system and I was to look at all the blocks in the disk, I could divide that blocks into either their data blocks which means they are blocks that hold the contents of the files that the user is actually writing to. And there are other blocks which are which I can call the metadata blocks.

Metadata blocks are blocks like the super block, the inode blocks, the free bitmap blocks, the directory the blocks that contain the contents of the directory even they are metadata blocks, right. So, all the blocks that are associated with the semantics of the file system are metadata blocks. All the blocks that are not associated with the semantics of the file system are data blocks, right.

So, for example, the contents that blocks that are stored in the contents of a file have nothing to do with the semantics of the file system, right. So, they are metadata blocks and everything else will be metadata blocks. And so far, as we have discussed the logging file system, we are basically saying that we log everything, we logged both data

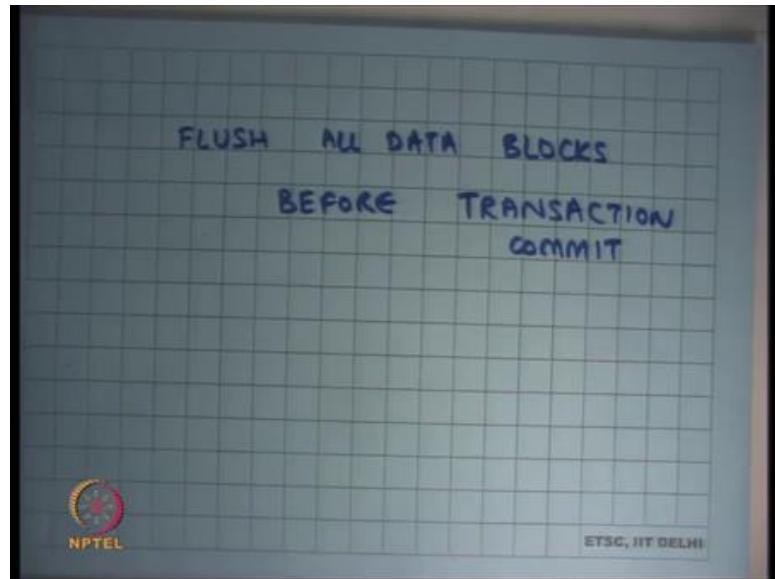
blocks and metadata blocks. But you know notice that the logging data blocks is not necessary for making sure that your file system remains consistent, alright.

So, and the other thing is that logging all data blocks may be expensive, if you can avoid logging data blocks and only log metadata blocks and you know do not. So, you avoid the 2x writes on the data blocks you only do 2x writes on the metadata blocks that is likely to be more efficient than logging everything, right. So, that is actually the default mode. So, ext3 has 3 two modes, ordered mode which says log only metadata blocks and journaled mode, journaled we had said everything, and the ordered mode is the default mode.

So, most of us most people use the ordered mode. And the order mode you only do this logging or atomicity maintenance for metadata blocks, but you need to be careful because imagine if the inode has been committed, but the data block that the inode is pointing to has not been committed to the disk then you will have a dangling pointer in your inode, right.

So, in the ordered mode the invariant that is followed is that you will flush or write to disk all the data blocks belonging to the transaction before committing the transaction which contains the log of the metadata blocks, right. So, in the ordered mode you basically ensure that all the data blocks are written to disk before you commit the current transaction that is all. So, instead of logging the data blocks you just make sure that there is an ordering relationship between the data blocks and the metadata blocks. And so, the ordering relationship is something like this.

(Refer Slide Time: 09:33)



Flush all data blocks before commit transaction commit. So, you flush all the data blocks before you do a transaction commit. Notice that am in this ext in this ordered mode of journaled ext3 I am using both ideas from the logging idea and ideas from the ordering method, right. I am saying, I will flush all the data blocks before I commit the transaction. So, let us say let us see what happens.

If I flush all the data blocks and I have not yet committed the transaction and there is a failure what can happen. Well, you have just updated some data blocks on the disk, but you know you have not really updated the inode corresponding to those data blocks. So, there is there is really no problem. So, there is no inodes will probably be point into the old data blocks or some data blocks have been updated some have not been updated, but the file system remains consistent.

The contents of the file may have changed partially, but the file system itself remains consistent, right. So, it is possible that some know the user wrote 10 blocks only 5 blocks have been written to disk on a crash and 5 have not, but the file system itself is consistent, right. So, that is important thing.

So, the difference between the ordered mode and the journaled mode and the ext3 is ordered mode is not concerned a concerned about atomicity of full operations atom, ordered mode is just concerned about atomicity of all the metadata operations to ensure that the file system remains consistent. It does not care about the user side or the users

data. So, users really left to his own on his own devices to basically make sure that his logic remains correct, right. So, that is one thing, ok.

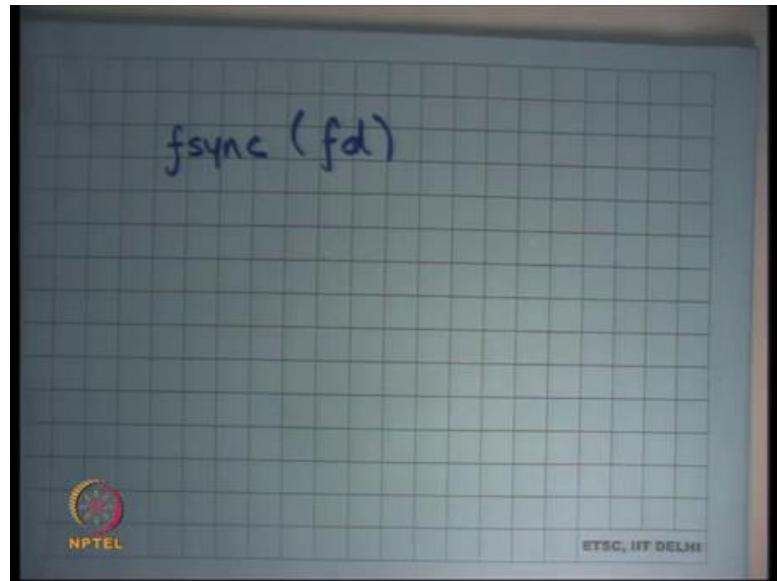
So, now, let us talk about how does the user. So, firstly, you know I hope we are all convinced that using this method we can be sure that the file system remains consistent. There are a few things one needs to be careful about you know, a few detail details about the ext3 ordered mode. For example, you know if you let us say delete a block you cannot use it in a free list.

So, before, so you cannot recycle blocks before the transaction commit because otherwise there can be inconsistencies. So, there are some details that I am skipping over this over, but basically you know at the high level its basically saying that you will flush the data blocks and you will know and you will not provide any consistency on the data blocks, but you will make sure that all the metadata blocks remain consistency, ok.

So, how does the user, I mean this gives no guarantee to the user. The user is doing something how does he know whether you know what he has done has been written to disk or not, right. So, for example, I have given you an example of an ATM machine let us say your ATM program was running on top of an operating system and the operating the ATM program said you know I want I want to make this transaction and he debits my account with some money and gives me the cache.

But the debit has not actually been reflected on disk; so, what is the device that the user has to make sure that the thing has, has been committed to disk.

(Refer Slide Time: 13:03)



And so, there is a system called fsync, right or you know a different operating systems can have different versions of this. And you can say you know I want to flush there is the contents the data blocks of this file to the to the disk, right. How will the file system implement something like fsync? How will ext 3 implement fsync let us say? In the journaled mode.

So, there are two modes in ext3 there is a journaled mode and there is an ordered mode. So, let us say in the journaled mode if the user calls fsync what should the operating system do before it returns from the system calls. It should commit the current transaction, it should commit the current transaction and that is it, does it need to wait for the transaction to get applied to the file system. Tree, before it returns from fsync.

Student: Yes.

How many say yes? 1, 2. How many say no? 2. Everybody else is undecided, ok. None of the above, alright. So, an answer is you do not need to wait for it to apply, right commit is enough. Recall that the semantics of the file system is that you know, anything that is been committed is as good as done. It is just a matter of you know apply syncing the state to the tree, now that is just something that and that is you know you can consider it as an optimization. So, that the next time you are looking through the tree you do not have to go to the log to look at the latest contents you know you can just do it

from the from the tree itself, ok. So, as soon as you commit the transaction it is as good as it is done on the disk and it will persist across powerfulness, right.

So, when you do an fsync the operating system just in the journaled mode just commits the current ongoing transaction and so, closes the current transaction and commits it before it returns.

Student: Sir, in case of recovery likes the power failure occurred in this stage after fsync, sir we should want that the tree should be write?

So, let us say if there is a power failure after fsync and you know you recover then you the question is should not you should not you desire, should not you require that the tree is correct well not really, right. I mean at recovery time you will just you know you know that there is a log. So, the tree could be in.

Student: already seen committed, right.

So, the log is showing committed, but the log has not been freed. Yes. Recall that the ext 3 file system first commits the log, after the commit it starts applying the log to the file system tree. If there is a failure in the middle or at the beginning of this application, it is ok, you know you have not freed the transaction you have just committed the transaction you will free it after the transaction has been applied completely to the file system disk, at that point you will free the transaction. So, in the journaled mode it is enough to basically commit the transaction and then return.

So, but you know an fsync is a relatively slow calls as you can imagine because an fsync cannot return without actually making a disk access and that is that is expected, right, unless of course, there were no dirty blocks in your block cache in your buffer cache. On the other hand, most other operating system most other system calls which is accessing the file system do not actually need to go to the disk because we are using it right back cache for the buffer, right. So, the fsync is a device for the user to basically make sure that that things are getting flushed to disk.

So, what happens if the user is using fsync too often? System becomes really slow you are write back cache becomes completely ineffective, you basically come you know reserve you basically look start looking like right through cache, you know let us say he

is doing fsync after every system call, it is basically like saying that every system call becomes one transaction. So, this system becomes really slow, right. And it becomes even slower for something like a transaction from, something like a logging system like ext3, right ext3 logging was actually heavily relying on the fact that you are batching lots of writes. They are doing 2x writes, but because you are batching them the performance is not really 2x is lower, right.

But if you as a user is actually calling lots of fsync you are actually 2x lower, you are writing it to the log and then you are applying them to the system to the tree, ok. Let us see what about the ordered mode. So, if I am using the ext 3 ordered mode and if the user calls fsync what do I need to do? I need to flush all the data blocks and I need to commit the transaction which contains all the metadata box, right, that is all, ok.

So, what are the some types of applications that require fsync? Why that require the data beyond the disk before you know some you do something? And most of these; so, such applications are usually what are called transactional applications or transactions, where you know, and an ATM transaction is an example of that, or a bank transaction is an example of that.

You basically want that you know, you basically have committed to the disk before you actually display to the user that your amount has changed or you actually dispense cache to the user etcetera, right. So, you would want something like that. And how are such applications typically implemented? These applications are typically implemented by using what is called a database, right.

So, database basically maintains all this state you know who has how much money etcetera and database is supposed to give you guarantees that you know if you have said something, you have asked the database to do something, it has actually done it on disk it has committed the whatever operations are you done by a commit in a transaction of a database you basically make sure basically mean that the data will persist across power failures.

So, if you are running a database on top of an operating system you know as an application, then the only way the database can provide you those guarantees is by calling the fsync call, right.

So, database is an example, a user space database or a database running as an application is an example of an application that makes lots of fsync calls and is likely to be very slow on something like this, right. And that is the reason you would typically implement a database as a standalone system without an operating system below it, right.

So, you would not implement a database on top of an operating system because you have two layers of management of disk. It is been, you know if the disk, if the data the database should optimize itself, so if the database had come direct access to the disk it would have been able to optimize these things much better, right as opposed to sitting on top of a file system like ext3, ok.

On the other hand, most other applications do not care about persistence for transactions or and so, they can run easily, and they do not need to call fsync calls and they perform excellently all on something like ext3.

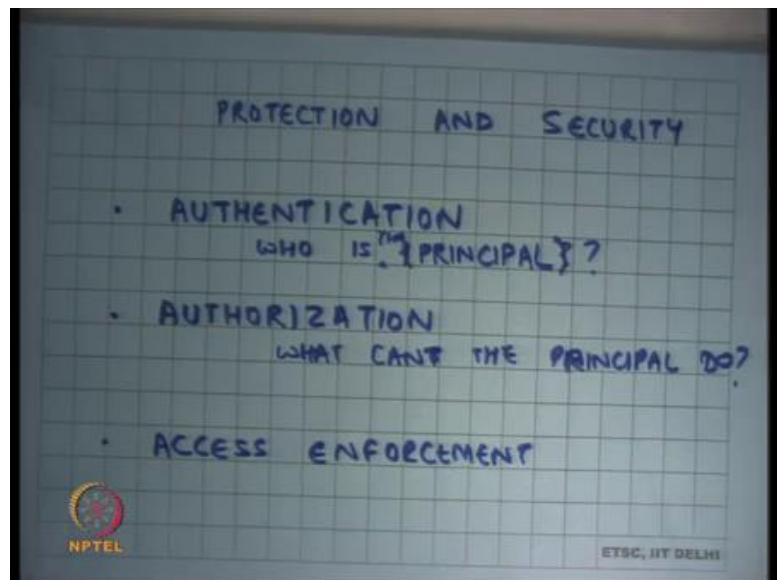
Student: Sir. So, the DBMS has the, it does not have those privilege levels to actually access the disk directly?

Does the DBMS have the privilege levels to access the disk directly? Well, I mean, so I said there are two ways to implement a database management system DBMS, one is you implement it as an application on top of an operating system in which case it cannot access anything. The other way to implement a DBMS is to implement it as a kernel itself or within the kernel, right in which case it has all the privileges, right.

And so, what I am saying is you know here is a very good example why you would want to implement it in the kernel as opposed to doing it at the application level because you do it at the application level, you have to go through the operating system interfaces. And, the operating system is playing tricks underneath and then you want certain guarantees out of it and so, the total performance sum of performance is really not what you would like it to be.

On the other hand, if the database had complete direct access to raw disk it would have been able to optimize these things much better, ok. So, that is you know I will finish. I will wrap up the topic on file systems with this discussion and moving on to the next topic protection and security, right.

(Refer Slide Time: 21:17)



So, we all know that we need you know we need lots of security or protection guarantees from an operating system. Your files on the shared infrastructure like the shared lab that we have should not be readable by other people, should not be writable by other people and so on. And so, you have a notion of these are you know these are my files, these are your files, you also have firstly, you have notion of identities. So, they are different identities each of us has a different identity in the system, right.

So, protection and security are consistent, is made up of 3 things authentication which is a method of saying who is who, right. So, our login ids or you know some other names basically say that who is who is behind this action, ok. So, we associate we have some notion of entities and we basically have some way of ensuring that this person is behind this action or this entity is behind this action. This entity is also called principle, alright.

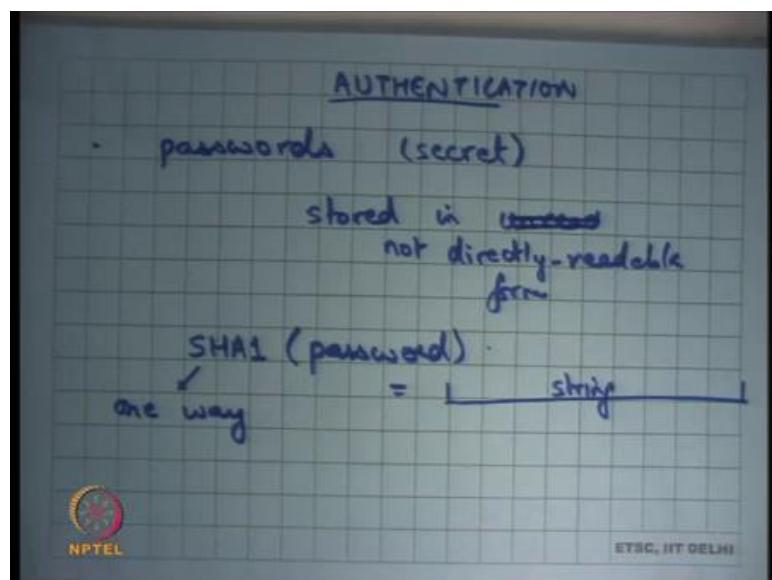
So, authentication identifies the responsible principles behind each action. Then there is authorization. So, this says who is the principal behind; let us just say this authorization is what can the principal do, right. So, basically says that user xyz can access files a b c in read mode and access files d e f in write mode and so on, right. User route can access all files in a b c in read write mode for example. So, these are all examples of who can do what, right. So, that is authorization, who is authorized to do what actions.

And, then these access enforce enforcement. So, you have a you have a method to do authentication, who is behind this action. You have a method to do you have a method to

do authorization, what can this principle do and then there is an access enforcement which ensures that only legal things are possible which means only things that a principle is allowed to do is as possible, right. So, these are these are these are the 3 sort of aspects of prediction and security.

And, if there is a flaw in any one of these then there is a security attack. So, let us say if there is a flaw in the authorization and you can impersonate as me, then you can do anything you will you like. If you can make you can add a flaw to the authorization table and say you know I can do a something that you are not supposed to do then there is a problem. And, thirdly even if you have all these things correctly, but if the access enforcement is not correct then you know there is a problem, ok.

(Refer Slide Time: 24:47)



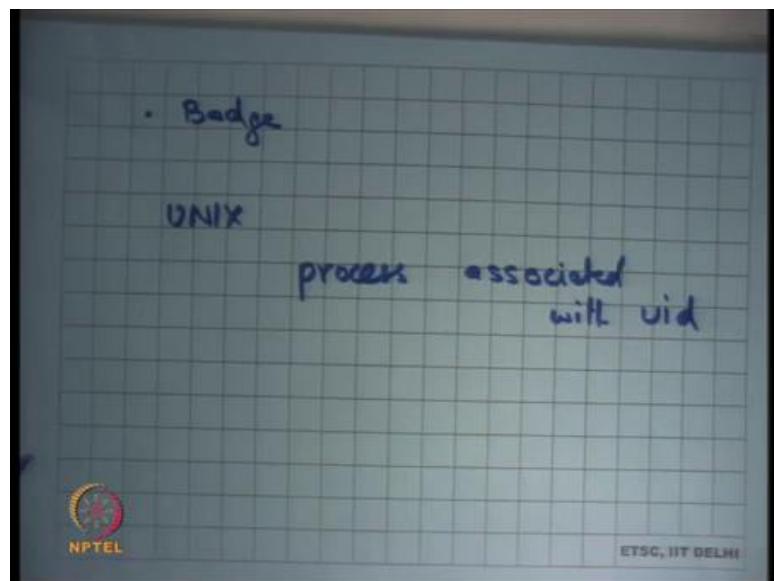
So, let us look at you know how these things are done. Authentication you know let us say passwords. You all know what our passwords, they are relatively weak way of establishing user identity usually stored. So firstly, they are secret and stored in unreadable form or not directly readable form, right. So, the way passwords work is that you store, so let us say I have a password on a system then you will use a one-way hash.

Let us say there is a there is a famous for there is the popular one-way hash called SHA 1. And you will apply the SHA 1 my password and you will get a large string, ok. And this is a string that you will store on your file system, ok. And this function is what is called a one-way function.

So, by one way it basically means that it is very easy to get this string from this password but given the string it is very hard to recover the password to know what the password is, ok. So, it is very hard to recover. So, even if you have the string you will not be able to you know guess the password and so, even if you have the string you will never be able to log in to the system because you will not be able to guess the password. So, that is typically how you will store these things.

The best attack you know assuming that you know our cryptographical algorithms are correct, our axioms in the science in computational theory are correct, then the only way to guess or to break this password based authentication is through brute force attacks where you basically go through all possible passwords up to a certain length and you basically check and so, this and so to the way to make this secure is to make sure that your passwords are long enough, so that brute force attacks are relatively ineffective, right. So, that is basically that is pretty much mostly how authorization is done. Alternate form of authorization is badges, right.

(Refer Slide Time: 27:19)



So, everybody has a badge a badge could be something you know a card that you are using to authorize yourself or it could be a USB, USB device that you have to plug in etcetera. It is the advantage of something like that it does not have to be kept secret should not be forceable or copiable, but you know if its stolen then at least the owner

know that its stolen and can register a complaint etcetera and or change the change the authentication mechanism.

If you are using badges you know there is an interesting paradox that the badges should be easy to make, cheap to make so that you can distribute lots of badges to lots of people, but should be very hard to duplicate badges. So, it should not be possible that if you have a badge and I just get access to it for few for some time I should not be able to duplicate it.

So, you know there should be some technology there that ensures that is not possible to duplicate it and yet it should be cheap to sort of manufacture, right. So, once authentication is complete you basically all the actions that are performed are basically done with that principles id. What does this mean in the context of an operating system? It basically means that all the processes run with that user ID, right

So, basically on UNIX every process associated with uid, right. So, recall that process control block that we had for every process. So, apart from all the other Meta information and other information that you will have is basically who is the user id or the particular process, right and that is; so, whichever process is running you always know who is the uid of that particular process.

(Refer Slide Time: 28:59)

AUTHORIZATION
object

PRINCIPALS

User A	File A	device B	memory
User B		Read + write	Read
:			
:			

Access Matrix

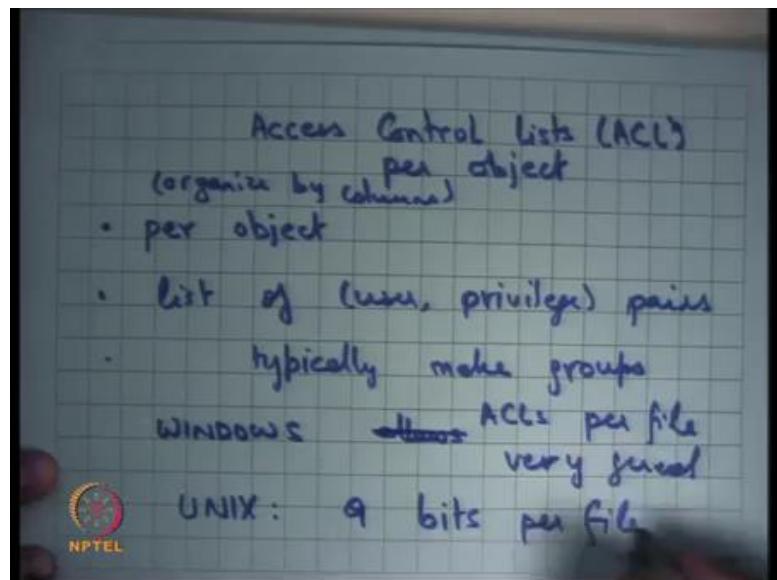
So, let us see authorization. So, who is authorized to do what? First let us look at the you know what does authorization mean in the more general sense. Authorization is nothing but a matrix, where let us say you have one row of principles. So, these are principles. Let us say this is user A, user B and so on and on the columns, you basically have objects.

So, for example, may file A, device B, memory region C and so on, right. These are all examples of objects that you have, and this authorization table basically says that user B can access memory region C or not, right and in what way. So, let us say read or read plus write etcetera, right. So, that is basically not a on a logical level that is what an authorization matrix looks like and this is also called an access matrix.

So, this is a logical representation of who can access what, but different things different authorizations are implemented. So, this is the logical authorization mechanism, but different authorizations or different authorization for different types of objects are implemented in different ways. For example, authorization for memory is implemented using virtual memory subsystems like page tables and segmentation, right.

So, you are whether you can access this particular region of memory or not is basically authorized, the authorization is implemented using some kind of a page table. Authorization for a file is implemented in a different way, ok. So, we are going to discuss that very soon. So, in practice you do not store the full access matrix there are two ways to store the access matrix either you store what are called access control lists Or ACL per object.

(Refer Slide Time: 31:17)



So, here you basically saying that I will organize things by column. So, who all can access memory C or who all can access file A.? I am going to have with file A I am going to have a list of all the users who are allowed to access it and in what way are they allowed to access it, whether read or write or execute or whatever else, ok. So, that is an access control list. So, per object organized by columns, per object, list of user privilege pairs, alright.

So, let us now this list can become large. So, typically, typically make groups and indicate that this resource is accessed by this group. So, you organize users into some groups, and you say that this file is accessible by this group, so all the users in that group can access it without those trouble etcetera. So, let us look at it in practice Windows allows, so Windows implements ACLs per file very general. So, on Windows you can go to you know the properties of a file and specify a long list of which user can do what, right.

So, here is an example on reading system that allows very general ACLs ah. Something like UNIX has 9 bits per file, right and we are going to see what these 9 bits are.

(Refer Slide Time: 33:25)

	owner	group	others
read	1	1	0
write	1	0	0
execute	0	0	1

"root"

So, UNIX ACLs; so, firstly, it divides, so let us say, so I am talking about a file it divides it into it divides all principles into 3 types owner, group and others, and read, write execute, right. So, this becomes, so there are 9 bits there could be 0 or 1. So, that is basically the UNIX ACL. Every file is associated with one owner and one group, right.

So, every file on the UNIX file system will say you know the owner is user A and the group is group B and you know the creator of that file can or the owner of the file can choose which group this particular file belongs to or is owned by. And then you can have ACLs of this type which basically say that this particular user is allowed to and so, the owner can do these things to this for example, the owner can read and write, but not execute this file you may want to say that.

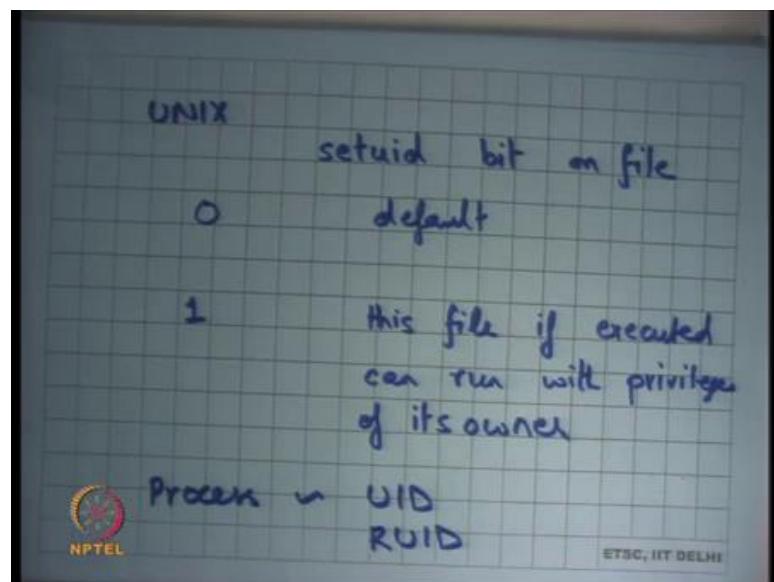
The group can read the file, but not write to it and others everybody else should not be able to either read or write. So, let us say here is an example. This is 1, this is 1, this is 0, this is 1, 0, 0 and this is 0, 0, 0, right that is 9 bits to ensure to basically implement ACLs. So, clearly this has less, there is less general generality than a complete list that you had on windows.

There is another thing about UNIX, there is a special user called root and root user is allowed to do everything to every file, right. So, for the root of these this access matrix is not these 9 bits are immaterial; the root user can do anything to any file for example. Yes.

Student: Sir, as in the group the group permission should be a subset of owner permissions?

Should the group permissions be a subset of owner permissions? No, no. See, it is a full 9 bits, so you can do whatever you like, ok. So, that is basically you know that makes it. So, the advantage of doing this is that you have a bound on the size of information meta information that you have you know you will store you can store this information, you can hope to store this information in the inode of the file as opposed to an unbounded list of user privilege pairs which will be very difficult to maintain, alright.

(Refer Slide Time: 36:11)



Apart from this, so the UNIX also provides an interesting thing which is a setuid bit on the file, right. This basically says so, with, so by default at 0, so you know it is the default behavior whatever you expect, but let us say this file is 1 then this file is executed can run with privileges of its owner, ok. So, every file also has one bit which is called the set uid bit.

If this bit is set then you can run when if some if let us say I have a file in my home my directory, but I have set the set uid bit to 1 and I am basically allowing anybody else to run this program and with my privileges. So, for example, this program if run, so recall that I said that any process that you that user x invokes will run with the privileges of user x, but if this process is exiting a file that had the set uid bit set or 1 then this program can also execute with the privileges of the owner, ok.

So, why is this required? Let us first understand what is the application, why do you think this kind of thing is required.

Student: (Refer Time: 37:47) this file is executable file leads to execute some other executable files.

If this executable file leads to execute some other executable files so.

Student: It is like it should have net privileges of the.

Well, I mean let us say all the other ones are also executable by others, so you know then there is no problem. Why do you need this? Are you; I think all of you are using an executive call set uid you know through the course of this project of this course? Can you think of it?

Student: (Refer Time: 38:25).

Ha.

Student: (Refer Time: 38:27).

Yeah, all the scripts that you are using on policy are actually set uid files, right. So, if you want to submit your assignment, so what I have done is I have a script which is living in my, I have a file which is living in my home directory whose owner is you know whose owner is me. And, but when you run it you can actually have the privileges that process has the privileges to write into my home directory, ok. For example, it copies files from your home directory into my home directory to make the submission.

So, how can the process that you are invoking has the privileges to write to my home directory? Right. I said all the processes that you invoke will run with your privileges and you do not have privilege to write to my home directory, right. So, the way to do that is basically to make this file to be executed with my privileges, right. So, that is what the set uid bit allows you to do. The set uid bit allows the owner to say that this particular logic you know I have tested this particular logic and this is safe and I allow other people to run this particular piece of logic with my permission, so that they can access resources that I could access. Yes.

Student: Sir, is that similar to just saying that for example, part of ux is my privilege then I will set the others privileges also to (Refer Time: 39:47). So, is not that similar that my owner privileges are also what yeah those?

Is not it similar to saying that you know I have an execute bit set to a file? Now, what is a difference between a set uid bit; I am just saying that this file is executable by others.

Student: (Refer Time: 40:05).

Right. If I just say that this file is executable by you will be able to execute it, but this process that will execute out of this you know out of your invocation will execute with your privileges not with my privileges and so, it will never be able to touch my files or write to my directories, right. So, set uid allows you to not just execute the file, but also execute it with the privileges of the owner of that file which also means that if there is a bug in this program then you can trick this program into you know doing things that I would not want you to do for example, right, ok.

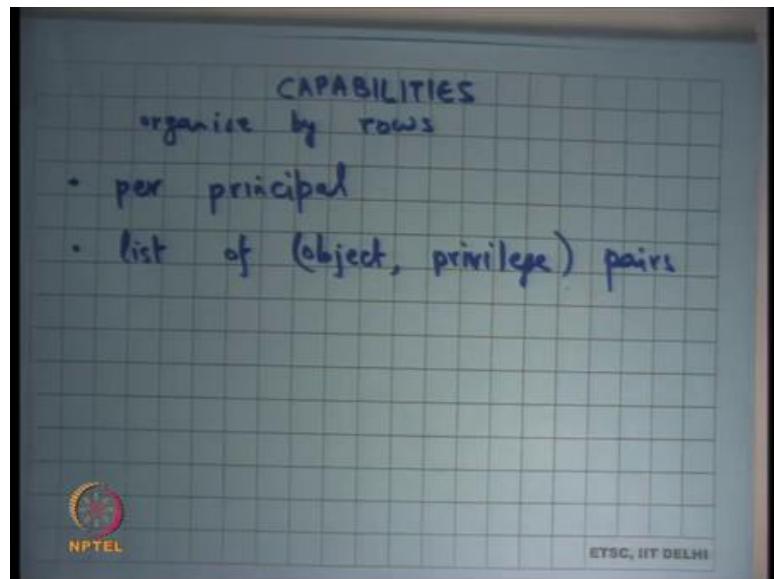
So, also this you know typically when you do these things for example, the submit script that we have will not just is not as capable of writing to my directory, it is also capable of reading from your directory irrespective of what permissions you have for your directory. So, if your if your permissions for your directory are saying that only I can read to this directory, yet this particular process that is you know that is that can run as my privileges can actually access your particular directory. So, this process can either can switch between your privileges and my privileges, right.

So, how does how is this done? Well, I said every process has associated with a uid which says you know who owns this. Actually, every processes two identifiers, one is called the uid and another is called the real uid, right. And so, a process can switch between its uid and its real uid at will, right.

So, they are using system calls, it can basically say that now I want to run with the privileges of my real uid and then I can say I want to now run with the privilege of uid. By default of the set uid bit of a process is not set then you then real uid and uid are equal to your uid who has invoking it, but if its if set then you know real uid will be your uid and the uid would be let us say the owners uid and, so the process has flexibility to switch between the two.

So, when I want to read from when this process wants to read from your directory it should switch to your uid, when he wants to switch to my directory, he wants to he should switch to my uid and so, right. So, that basically allows him to do this, ok, right.

(Refer Slide Time: 42:49)



The other way to implement access control are capabilities, what is called capabilities. So, we looked at access control lists. The other way to do access control is called capabilities where you organized by rows, right. So, where basically you organized by rows, the access matrix; so, per principal, list of you know list of object privilege pair let us say. So, user x can access all these files or user x is allowed to access. So, you know you store this information in per user as supposed to per file that the other way to implement this, ok.

So, let us see what are some examples of capabilities that we know about. Well, here is an example file descriptors. So, every process has the set of file descriptors that basically say that these are the files that I can access, or I have already opened. So, I can you know I can access it in xyz way. And so basically a process can be thought of as a principle and all these file or these pointers to file blocks or structures that that represent open files can be thought of as objects and you basically storing file descriptor table that is your capability list, what are the capabilities that this process has.

So, one some advantages of capabilities are that let us say I spawn a new process. So, it is very easy to delegate. So, if I want, I have some capabilities I want to give you those

capabilities I just give you my table and you know you have the same level of capabilities. For example, forking the process involves giving my capabilities to my child process, alright. So, that is that similarly you know that is a memory management, so virtual memory. So, you basically say that you know what all areas of memory are you allowed to access this is kept per process. It is a you know it is a capability list per process.

In general, the advantage of capabilities is that it is very easy to delegate. So, you can you know giving, if I have some capability, I can give you the same capability by just giving you access to the same table basically, right. The other thing is capabilities are also a little more secure in the sense that only if I have a capability can, I even name the system, right, name the object, right. So, capabilities are also used as a way of naming the object. For example, if I you know if I have an, so for example, virtual memory, so I can name this particular location because I have a capability of accessing that particular location.

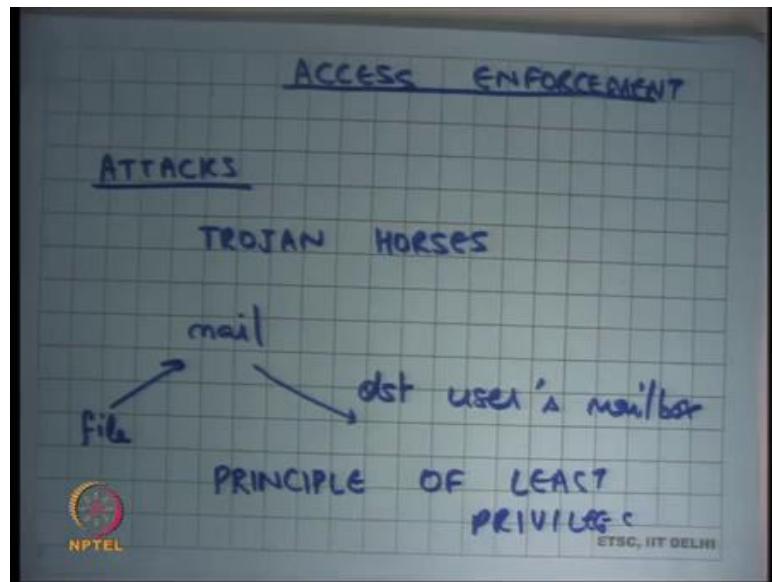
Another process who does not have the capability of accessing this location will not even have the name on in or the way to access that location. So, in general you know capabilities allow private namespaces, so basically you have a namespace for this process or namespace for this user and a name space for that user and this user does not even know that those files exist you know all those resources or those objects exist which are in your capability list. Whereas, if you have access control lists the entire namespace is public, ok.

So, capability systems discard its visibility, namespaces are private by default and so, in there they also thought to be more secure than access control list in that sense and there have been experimental systems that have tried to implement use capabilities for file system instead of access control lists, but you know there they tend to be a little clumsy for file systems whether I used for other things like you know virtual memory systems etcetera. They are also used in distributed systems a lot and capabilities can are also implemented using cryptographic means, right. So, for example, a capability could involve a signed key or a signed certificate.

So, let us say I got a you know if I have if I want to authenticate to somebody I can use or I can I can use a certificate from sub; so, if I want to prove my authorization to access

a resource I can show a certificate for somebody else to show say that I can access this particular resource, right. So, that certificate, so crypto capabilities can also be implemented using cryptographic means basically, ok. Finally, enforcing access and for a kernel, so what does access enforcement mean.

(Refer Slide Time: 47:25)



So, we have seen ways of saying that here is, here is how you authenticate. You authenticate using passwords or badges or some manual biometrics or whatever. Then you have some way of authorizing which includes access control lists or capabilities access control lists or capabilities and you know depending on. And, there is a there is a trade off on how much space you want to take to represent this stuff and how much generality you want, and we have seen some an interesting example of UNIX versus windows. And then you have some way of ensuring that these things are obeyed, right.

So, which this basically means that some part of the system should be responsible for enforcing this and this part of the system should have total power on everything, ok. So, there has to be some part of the system which has total power and its basically and this part of the system is checking and saying yes or no to actual controls and for an operating system this part of the system is the OS kernel basically, which can which can which has total control and it is going to do authentication and authorization.

So, let us look at some attacks. So, why do you know let us say what are the what are some kinds of attacks that happened, what does what does the security attack mean. So,

we keep hearing that you know xyz, there is xyz vulnerability there is this attack and that attack, what do these things really mean. Basically, an attack means that a user has firstly, it basically means that a principal has been able to do something that he was not authorized to do and the most common way of doing an attack is basically what a Trojan horses, right.

Trojan horses are basically, so from derived from the mythical tale. A Trojan horse basically means that a useful program that was supposed to do something legal and meaningful is subverted to do and had privileges of, had higher privileges than what you have is subverted to do something that it was not supposed to do, right. So, that is what a Trojan horse mean. So, let us see where all, what are some ways of implementing Trojan horses.

Well, let us you know let us take extreme case, if you could potentially subvert the kernel you know that is the ultimate Trojan horse that you can have because that has the ultimate power and if you can subvert that the access control enforcement engine itself then you have basically done, you can you can bypass everything else. Similarly, you know if you could subvert program that has a set uid bit set then you can get control over that particular programs resources.

Many programs you know it especially in the older days, many programs in a systems run with set uid bit set and owner as root, ok. So, which basically means that this program is going to execute as the root owner, root can will execute with root privileges, but anybody can invoke it, right. So, let us you know let me give you some example. So, let us say you know there is a program called mail, right that I can give some file to it and it will copy it to the destination user, so mailbox, alright.

So, let us say there is a program that I provide in my system that is called mail that says, basically I you know that takes an argument which is some text or some file which contains some data and I say I want to mail this program to this particular principle and what this program is going to do is copy that that data to that person's mailbox. So, let us say this is the program. So, this particular program in the older days would run using root privileges because this program, so this program will be a program which will be owned by the root user and we will have the set uid bit set because this program requires the capability.

Because, I can name anybody in my in the in my destination in the may recipient field and so, this you program should have the power to write to anybody's file mailbox. And so, this program for that reason needs to run with the root privileges, at least for some time and so, and with set uid bit set. So, if I could subvert such a program then I can basically get do anything that the root user could do.

And there are many types of attacks which are not the subject of this course that that can allow, so the many kinds of common bugs in programs that they were very common or you know 10 years back not so common today, so it is harder and harder to find bugs and programs that can be subverted, it was very common to do that 10 years back. And if you could if you could identify the bug you could exploit it to actually be as the program to behave in a certain way which the programmer did not expect it to be there.

In general, you know when you have programs of this type this you should follow what is called principle of least privilege which basically means that if there is a program that is running. So, if you have a large program let us say let us take the example of the mail program then it should try to run with the; so, it has this mail program has a set uid bit set. So, it can either run with root privileges or run with my privileges.

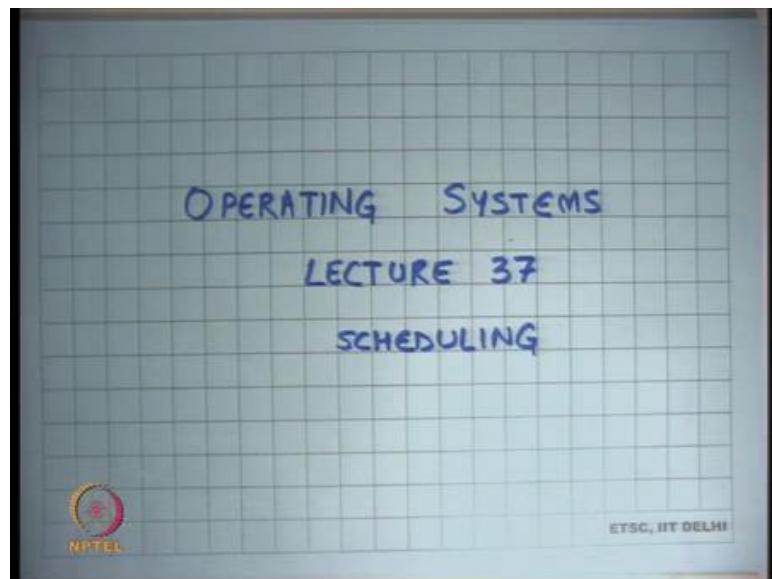
So, the program should run with the least privilege at any time. So, least privilege will be required at any time. So, for all the operations that do not require root privileges at that time the privilege the program should be running with my privileges. So, basically what that means is all the code that executes executed during that time if there is a bug in that code then there is no problem you know or you know actually there are fewer problems, as supposed to if you run all the time with the root privileges that is it, ok, alright.

So, with that let us finish. And we are going to discuss another topic next time which is called scheduling.

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

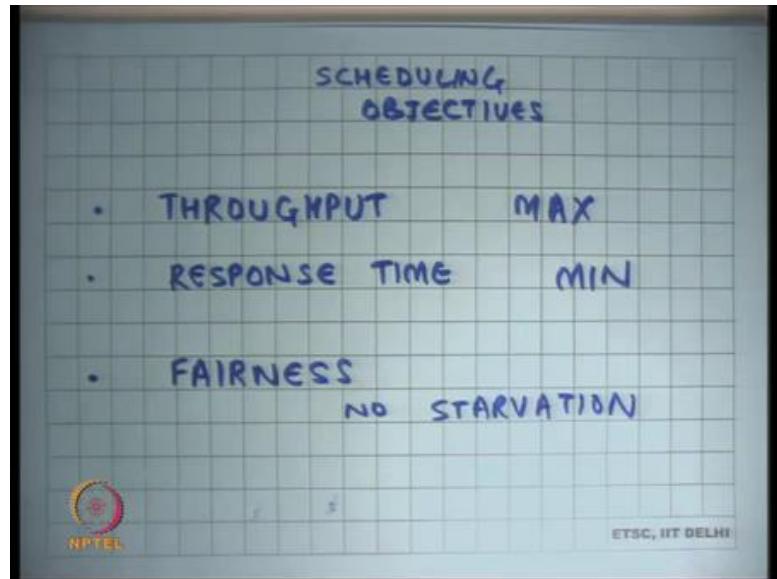
Lecture – 37
Scheduling Policies

(Refer Slide Time: 00:25)



Welcome to operating systems lecture-37 and today we going to talk about scheduling right. So, one of the central themes of an operating system is scheduling basically we have seen that an operating system runs many processes, handles many devices and maps processes to devices, and one of the central things that an operating system is really doing is scheduling right.

(Refer Slide Time: 00:47)



So, let us first understand what are the objectives of a scheduler? So, let us say scheduling objectives all right. So, let us say throughput, maximization, by this I mean that I want to act do the maximum possible work in the minimum possible time. So, there is a lot of work to be done, and I want to make sure that I finish as many jobs as possible in the minimum amount of time.

And so, I have these many resources and I want to judiciously use all these resources to do everything I can right. The other thing is minimize response time. This means that if as something needs to respond to something so if there is an into input output operation, then that response should be as fast as possible. For example, if the user presses a key, then he should see the key character getting printed on the screen as soon as possible right.

There should be some metric which says this is the response time, the response time is the time it took from between the pressing of the key and the display of the character on the monitor or you know the arrival of a packet, the arrival of a request on the over the network and the response or the reply over the network right.

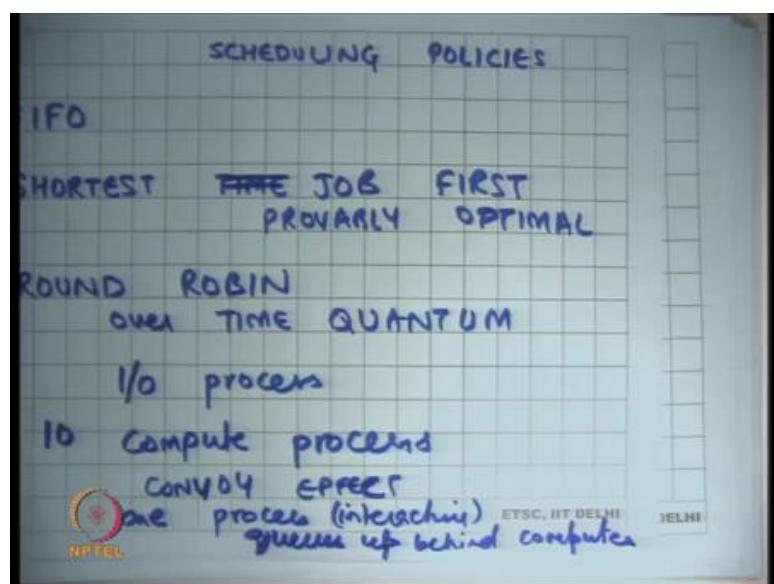
So, the user pressing a key and the character appearing on a monitor is a very; is a very easy constraint really, because humans perception time is very large you know. So, humans cannot perceive more than at more less than 1 millisecond granularity and computers are already as fast as nanoseconds, so that is not that big a deal. But if I talk

about network response times that becomes more of a more of an issue right, and it is not just network other IO devices also right.

And you are going to see that how you know these two requirements can conflict with each other. So, if you have maximum response, if you need maximum response time, it does not you know it may actually come at the expense of reduce throughput. And then you want some level of fairness right. So, in doing all this, you make you want to make sure that if there are many jobs that are running, there is some amount of fairness in the system which it should not be that if a job is giving you a lot of throughput, you just keep running that particular job and did not run any other job, so there should be some level of fairness.

In particular there should not be any starvation right. So, there should be no starvation. By starvation I mean there is some job that does not get to run at all or there is no guarantee on how frequently it gets to run right, so it can keep waiting for an unbounded amount of time that will be called starvation and this you know a scheduler should actually never allow something like starvation right.

(Refer Slide Time: 03:45)



So, those are some of our objectives then let us look at some types of scheduling policies all right. So, the simplest policy is first in first out or first come first out right that is your typical scheduling policy, you queue up the jobs in arrival order and you keep doing the jobs in the order that they arrived to you right. This is ok, but you know the process lens

can be actually very large you know, some process lens can be very small other process lens can be very large.

For example, if you run a compute bound process and the resource at your scheduling is the CPU, then the process compute bound process could probably take you know minutes and you do not you cannot afford to make other jobs wait for minutes right, so there is a there is an obvious problem with that.

Well, in the same vein there is something called shortest time completion first or shortest job first let us say. Here again instead of serving the jobs in the arrival order, you basically say here are all my jobs that I have come let us say I wait for some time and I collect some jobs, and I say what is the shortest job in this buffer and I picked that and as you do that and then I pick the next shortest job and so on all right.

So, with if I do that then you know arguably I will minimize the amount of response time for the you know the total average response time per job if I do that so this is you know provably optimal in terms of response time average response time per job, but once again it's too unfair to be actually practical right.

It is personally it you know you know the job even the shortest jobs could be very long in that which means that other people can have to wait. And secondly, if you know there is a lot of short jobs that are arriving all the time, then the lock job can just keep starving forever right, so these are all this is also not really very practical.

For something like a CPU, you could do something like round robin and define some kind of a time quantum right. So, recall that we said that there is some amount of there is a scheduling quantum and we define the scheduling quantum based on some matrix, one of the scheduling quantum that we sort of looked at was 10 milliseconds or 100 milliseconds.

How was the scheduling quantum decided? Firstly, the cost of doing a context switch should be very small with respect to the scheduling quantum, so that the overhead of scheduling is small number 1. And number 2, the scheduling quantum should be small enough such that so it cannot be too large if let us say the scheduling quantum was two seconds, then you know if I press a key and there was some job that is running currently,

then I will have to wait on average one second before I actually see that key to be displayed right.

So, whatever are my input output devices the response time should be proportional or roughly proportional to the scheduling quantum that you choose, right. So, if you are you know if you are worrying about human users, then human users need you know millisecond response time or 10 millisecond response times and so scheduling quantum should have some proportionality to the amount of response times that you are looking for or some relation with the amount of response times we are looking for.

As an interesting piece of fact, you know even in the 1970s the scheduling quantum used to be 10 milliseconds and even in today 2014 the scheduling quantum are still 10 milliseconds. Even though the computers have become really faster right, and then at that time the computers were 60 mega Hertz, today the computers are 2 giga Hertz so you know there is a 100x speed up roughly speaking. And so, what is so but the scheduling quantum have not changed, what is the reason.

Student: You are saying human response time.

Humans human response times have not changed right so good, all right all right, so that is round robin. But round robin completely ignores the type of the job, so you basically give all jobs completely equal you know it is completely fair and that may not be the best thing. For example, let us say there was an IO job IO process and there were as a compute process and let us say there were 10 compute processes all right.

So, now you are doing round robin between these compute processes and these IO and this one IO process and what you care about is not just your throughput, but also your response time right. Compute processes mean there is some you know long running computation that is going on. So, there is no IO in the compute process and so there is nothing like there is no notion of response time in the compute bound process, but for the IO bound process there is something called the response time.

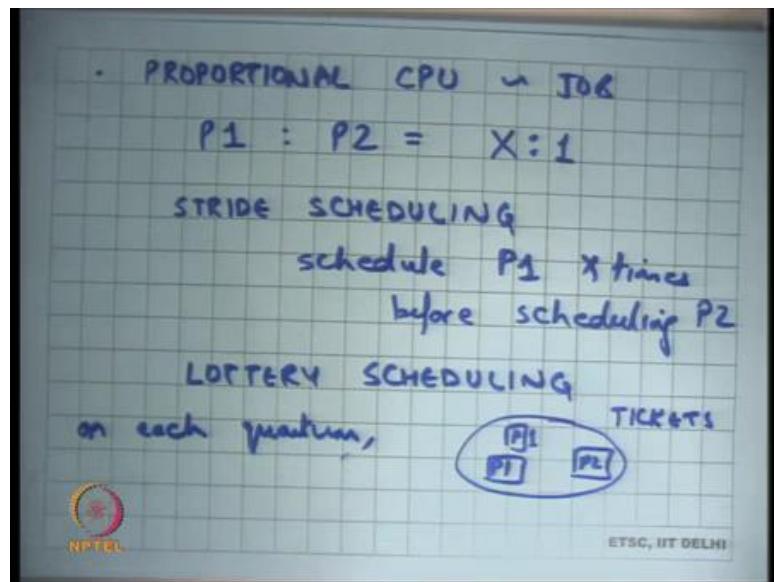
And so what you would want is that whenever the IO bound process is ready, you will want to run it first respective of how many compute bound processes are running at that point right, because if I press a key, let us say the IO bound process is your editor and it is you know it is currently waiting for a keyboard key press. And I press the key, now if I

just do pure round robin scheduling, then this IO bound processes process will have, may have to wait for 10 compute bound processes to finish or 10 scheduling quantum to elapse before I get to run right, so this is also called the convoy effect.

There one process interactive process queues up behind non-interactive processes or compute processes this is in computes, but compute processes, so that is not a good thing. And we will see how one can solve that. So, if I do a pure round robin scheduling, I can actually have a very poor response time. So, let us say I have a system that is running lots of compute bound jobs, let us say it is running simulations at the background that should not mean that if I press a key or if I login into the system, you know my the time it takes for the next prompt to show is really high right.

Even if there are lots of jobs running at the background, my system should appear interactive and there is the cost I pay for making the system interactive is really small right. So, it should both be doing the compute bound jobs and yet give me a very high level of interactivity and that is what I would like to have ok.

(Refer Slide Time: 10:32)



Also, round robin is just to fear you would probably want some level of proportional CPU depending on the job right. So, for example, some processes are more you know are more important than other processes, system level processes are more important than user level processes.

Let us say there is a process that is going to handle the disk, it is going to flush the disk, or it is responsible for doing something on which means lots of different user process, user level processes rely. And you want to say that the system level process should get more CPU than the user levels process on average, right.

So, you could say something like you know let us say process P1 is to process P2 is the you know the proportion should be X is to 1. And then you can just modify this round robin to basically do this X is to 1, proportional share scheduling. How can you do that? Well, one way to do that is what is called stride scheduling, where you schedule P1 x times before scheduling P2 all right, so that would not just a straightforward extension to round robin, you just add some proportionality to it and you basically say I am going to schedule this x times before I actually schedule.

The other way to do that is what is called lottery scheduling ok. And lottery scheduling basically is a probabilistic algorithm where you say that you know you have a pool of tickets, lottery tickets, and each ticket has a label which is let us say P1, P2, P1 and so on. And on each quantum, the see the OS draws one ticket at random from this pool.

And depending on whose ticket it is that is the CPU that gets to run right. So, it is a randomized algorithm it is a randomized version of straight scheduling, where you basically just pick up one random ticket from the pool, and just say that whoever comes is basically the one that gets to run. So, if I want to implement proportional shear scheduling, all I need to do is have x tickets for P1 and for everyone ticket of P2 and so on ok.

Student: Sir, we got how will larger scheduling high some of the corner (Refer Time: 13:09).

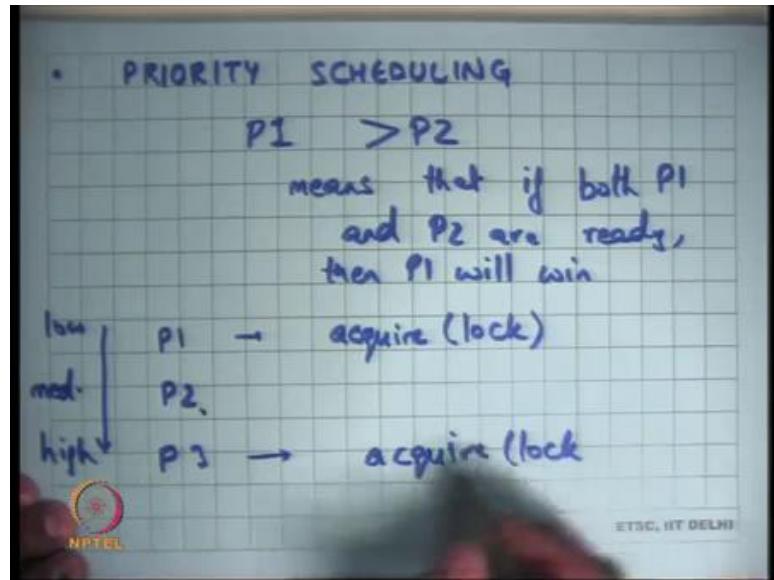
So.

Student: (Refer Time: 13:11) that its chance is not going to have.

Right, right, right. So, I am not solving the convoy effect right now, I am just talking about you know proportional share scheduling, and we are going to talk about what convoy effect very soon ok. The nice thing about these scheduling algorithms are they are tolerant to dynamic admission of processes and dynamic exit or the processes.

So, processes are coming in and going out, and that is completely right. So, if a process comes in for lottery for some for something like lottery scheduling, I will just add that many tickets to my pool, and you know it is a completely local express operation, and now you know it becomes proportional in that sense. A process exits, I remove the tickets for that particular process and that is it. Similarly, straight scheduling and round robin and etcetera ok.

(Refer Slide Time: 13:59)



But that is not, no, that does not solve our convoy effect, and then and so what we really need is some, some sort of priority scheduling all right, where the idea is that P1 has higher priority than P2 implies means that if both P1 and P2 are ready, then P1 will win all right.

So, in the proportional share, it was just an average thing you know basically a P1 will get x times more CPU than P2, but priority scheduling is a strict priority basically say is if P1 and P2 both are ready, then I will always give the CPU to P1 before I give the CPU to P2 right, so that is a strict priority scheduling.

So, some proper jobs have a higher priority than other jobs. Some examples of these are let us say interrupts or device if a if a device needs attention, I want to give it attention as soon as possible. So, interrupts you know we already know that interrupts have higher priority than other jobs.

So, let us say there is some job running, but somebody request an interrupt, the job gets preempted and the interrupt gets raised that is basically like that is basically saying that interrupts have higher priority or the device if the device needs attention the device will have higher priority than anything else all right. Or you could say you could solve the convoy effect by saying IO job bound has higher priority than compute bound job.

So, if there if the IO bound job is actually ready to run on the CPU which means there is some input that has been received from the keyboard. Now, it needs to be done on the CPU to actually do something, compute something and maybe display something on the monitor, then you know it should get higher priority.

So, even if there are hundred compute bound jobs, if there is IO bound job that is running, then it will get higher priority over the compute bound jobs all right. So, you could do something like that this.

Student: Sir, IO jobs should be a special case of device interrupts (Refer Time: 16:05).

Would not IO jobs be a special case of device interrupts, well, no not exactly, because a device interrupt may just mean that the device needs attention. And the device needs attention may mean that I just copy that character from there to my kernel buffer. And now you know I know that this character is meant for this particular process, so this process has become ready. And now that process becomes you know it is now the regular scheduling algorithm will pick up that process on the next scheduling quantum.

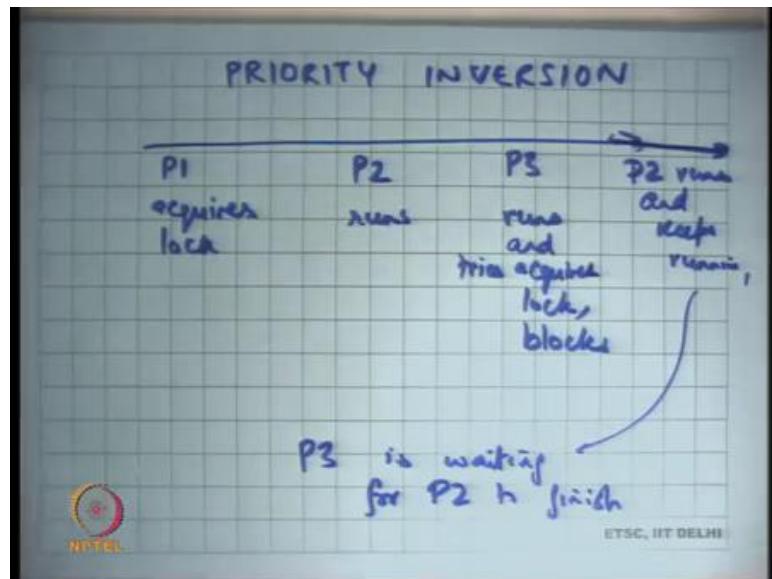
So, though both are examples of priority scheduling, but you know in different context. In one case, you are dealing with the device; in the other case you are dealing with the process that is handling that device all right. So, that is priority scheduling. You may also want priority scheduling for things like you know so some things are more much more important than other things, for example, real time operating systems.

So, you know if there is an operating system in the car, then somebody presses the break, then that should have the highest priority among all the other things like playing the music player etcetera right, so that is another thing example of a priority scheduler.

Now, you know when you do priority scheduling, you basically want strict priority. Strict priority is priority orders between these different jobs, but you know they are issues with that. So, let us say there were three processes P1, P2 and P3, and let us say the priority order had low priority to high priority, and let us say P2 is medium priority.

Now, let us say; let us say P1 acquires a lock or acquires any resource I am using with the lock as an example of a resource it requires a lock. And now P2 comes in and P2 gets to run. Then P3 comes in and P3 also tries to acquire the same lock or same resource that is it right.

(Refer Slide Time: 18:06)



So, the timeline is so this is my timeline. So, P1 acquires resource or lock; P2 runs; P3 runs, so P3 is higher priority, so P3 comes in; P3 starts running. And P3 it runs and tries to acquire the lock all right. But now because the lock is already held by P1, P3 will have to block; let us say and let us say it is a blocking lock, and so P3 blocks and tries to acquire a lock and blocks, and so P2 keeps running and keeps running.

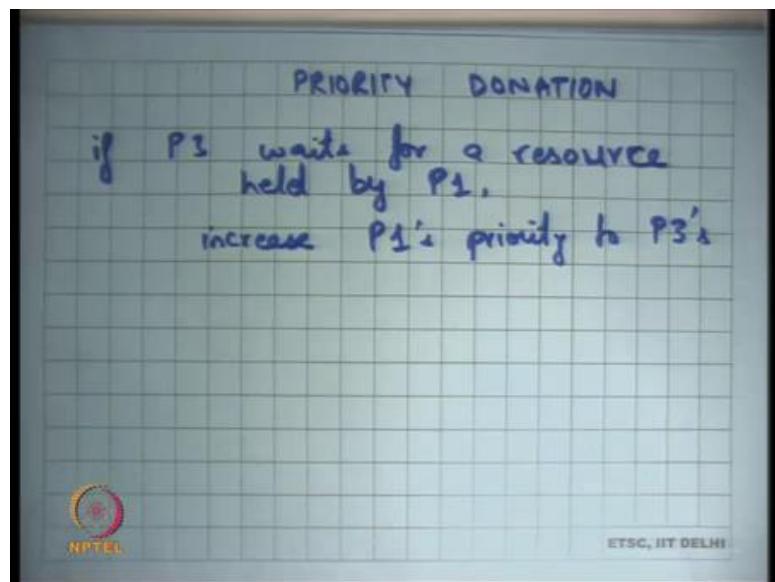
At this stage you have a situation, where P3 is waiting for P2 to finish right. And this violates the priority order. You always wanted that if P3 is ready to run, then P3 should get to run, but because P3 there is a dependency between P3 and the lower priority process, it is possible that P3 actually waits for a process that is lower priority than it, and this thing is called priority inversion right.

This problem is you know so this problem is basically once again the problem is that a higher priority process is waiting for a resource that is being held by a lower priority process, and but because the lower priority process never gets to run because there is a medium priority process that is already in the system.

And so, the medium priority process keeps getting to run, and so effectively what is happening is that the higher priority process is waiting for the medium priority process to finish, whereas that should not have happened. But the ideal thing to do here, so this is the this is what is called priority inversion. So, priorities have been inverted.

And so, the wait the ideal thing here would have been that the system if the system could figure out that the higher priority process is waiting for a resource that is being that is currently held by the lower priority process. And if that is the case then the priority of the lower priority process is bumped up to the priority of the waiting process, so that you know it gets higher priority than P2.

(Refer Slide Time: 20:29)



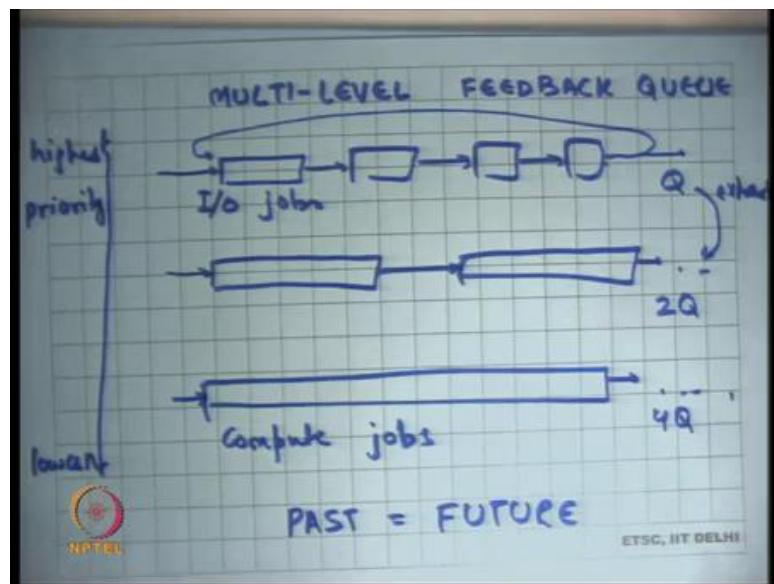
So, this problem can be solved by what is called priority donation. It says if P3 waits for a resource held by P1, increase P1's priority to P3's right. And so, you could solve this priority inversion problem by using priority donation. But but that is provided that this lock was something that was visible to the scheduler right.

So, I am assuming here that the scheduler or the operating system could figure out that P1 acquired a lock, and now P3 is waiting for a lock that is acquired by P1 that is only possible if the lock was a blocking lock or it was a lock that involved OS activity, for example, the lock involved a system call.

But if the OS has no idea about this dependency of between P3 and P1, then what is the worst thing that can happen that P1 holds a lock, P3 waits for P1 to complete, and P3 keeps waiting you know P3 let us say the lock was a spin lock, then P3 gets to run, and P3 just keeps running forever. And the lower priority low thread which is actually holding the lock does not get to run right.

So, that is the. So, you strict priorities can actually also result in deadlocks if the scheduler has no idea about the can as a fake to the dependencies or order resource dependencies between the different processes right.

(Refer Slide Time: 22:23)



So, using this priority idea what is used in operating systems is of what is popularly used in operating systems is what is called a multilevel feedback queue. So, what is this, you basically maintain multiple priority levels. So, this is highest priority, and let us say this is lowest priority.

And within each priority level you maintain the jobs in a round robin queue all right and so on. So, these are lower priority threads ok. You run the lower priority thread only if

so, you obey so by priority I mean you know the strict priority that we have discussed. So, you run a lower priority thread only if there is no higher priority thread available. So, you will run this only if all these other processes or jobs are not ready ok.

And so how and so basically what you would want to do is that all the IO jobs are at the highest priority, and all the compute jobs come at the lowest priority ok. So, basically what the system does is it organizes all these jobs into IO bound jobs and compute bound jobs, it gives highest priority to the IO bound jobs and serves all the jobs within the same priority in round robin order.

And then all the compute bound jobs are given lower priority than the IO bound jobs. And the and so if there is an IO bound job that is ready, then that will get served before there is a compute bound job that gets run. But then how does the system know what is an IO bound job and what is a compute bound job, number of?

Student: IOCs calls.

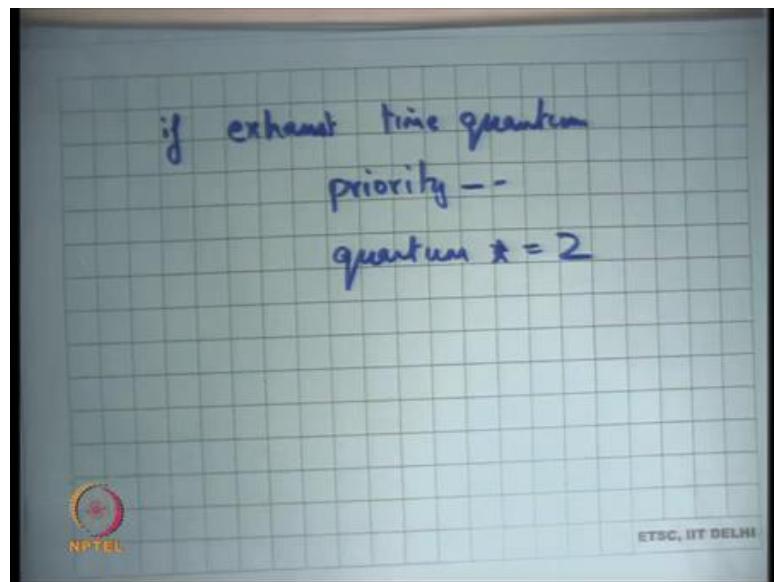
Number of IOCs calls ok. The number of IOCs calls, well, yes sort of. So, basically you just you know like as that everything else in the OS you will just use the past to approximate the future all right. So, basically say what has been the behavior of this application previously. And by IO, I basically mean something that actually blocks you know for whatever reason either IO or lock or something. And so, what you do is you start a job at the highest priority. And at each priority, you give lesser time quantum.

So, let us say there is a time quantum q here then you will give 2 Q here, and let us say 4 Q here. And you give it a quantum queue to run. If it exhausts the time quantum which means it actually does not block within the time quantum that you gave it, let us say you gave it 10 milliseconds to run and it did not add any exhausted the 10 milliseconds without blocking. If it is an IO bound job it is like it, it is likely that it will block before the 10 milliseconds require expire right. For example, it will go and say I want to read the next character from the keyboard.

So, if it is an IO bound job, it will not exhaust it is a time quantum. If it exhausts the time quantum it basically means that this jobs average length is longer than the time quantum that is given to it. And so, what you do is if you exhaust, then you move it to the lower priority queue right.

So, you start it from the highest priority queue and give it at one time quantum queue, if the job expires at time quantum, then you move it to a lower priority and also increase its time quantum right. If it does not exhaust a time quantum which means it blocks before it wanted it could expire time quantum you leave it where it was ok. So, this basically ensures.

(Refer Slide Time: 26:29)



And, you basically as you move the so the logic is if exhaust time quantum, then priority--, and quantum is multiplied by 2 right. So, you also multiply the quantum by 2 to make sure that you know the scheduling overhead is even less. So, it basically says that it seems like this job is there to you know take large chunks of CPU.

And so why to keep interrupting it if it needs a large chunk of CPU, you not just decrease its priority, but also give it higher amount of CPU. So, when it gets scheduled, it will actually get a larger chunk of CPU whenever it gets scheduled right.

And you maintain some levels to do this. So, you are, also it is not an unbounded list. So, you have a maximum bound on how much time quantum you going to have for a job ok. So, so question is that if there is a compute bound process that is running, and there was an interrupt that occurred in the middle of this compute bound process, then you know the interrupt will get served, and want that serving of the interrupt caused problems with the calculation of this quantum is that your question?

Student: No sir. If the (Refer Time: 27:42) occurring between and then some other process which was using a network being scheduled in it is process.

No. So, hold on. An interrupt will not, so and a network interrupt will not cause a scheduling behavior all right. Let us just; let us just work in this model that device interrupts do not cause scheduling changes right. Device interrupts are meant for device attentions. For example, a device interrupt has occurred let us say a network card created a interrupt, all it means is that the device needs some attention maybe it wants that I should copy some buffer from there to here, and then I will go resume the process that was already running that is all ok.

And the only where time that you do a scheduling switch is either when the process actually voluntarily yields the CPU or if there was a time and interrupt which caused the preemptive schedule switch ok. So, but what is going to do is that any process that is very IO bound. So, let us take an example let us say you are running you know gcc in your on your system and you are running an editor like vi.

So, vi is a compute bound pro is an IO bound process and gcc is a compute bound process. And so, let us say gc these are both long relatively long running processes. So, eventually what will happen is that gcc will come will have a lower priority, but a larger time quantum, and vi will have a higher priority and a smaller time quantum and that is exactly what you wanted right. vi does not need that much CPU, but it needs very highest attention spans; gcc needs lots of t CPU, it does not care about response times all right.

So, with this with that you can basically have some level of a you know you can have good levels of throughput and yet have acceptable levels of a response times. But let us say you know if I have something like this, then is not it possible that all the lower priority threads keep starving.

So, let us say there are lots of IO bound jobs that are waiting in the queue, then the compute bound job will actually never get to run right that is a so this is an this is a nice idea and principle, but you know one needs to do some modifications to make sure that there is no starvation possible. Anytime you have a strict priority system, starvation is always possible for the lower priorities processes. So, the way to deal with this is that

you basically if there is some process you basically keep increasing the priority of a process at some rate with time.

So, if a process has not been scheduled so for example, one way to implement this is on every schedule on every scheduling quantum if you do not use the process, implicitly you increase the probability of that process by some constant number what. So, all the processes that did not get chosen on that quantum, their process, their priority will get increased. And so eventually a process will get reach the highest priority and will get to run all right, so that way you deal with starvation.

Student: (Refer Time: 30:28) will be decreased right.

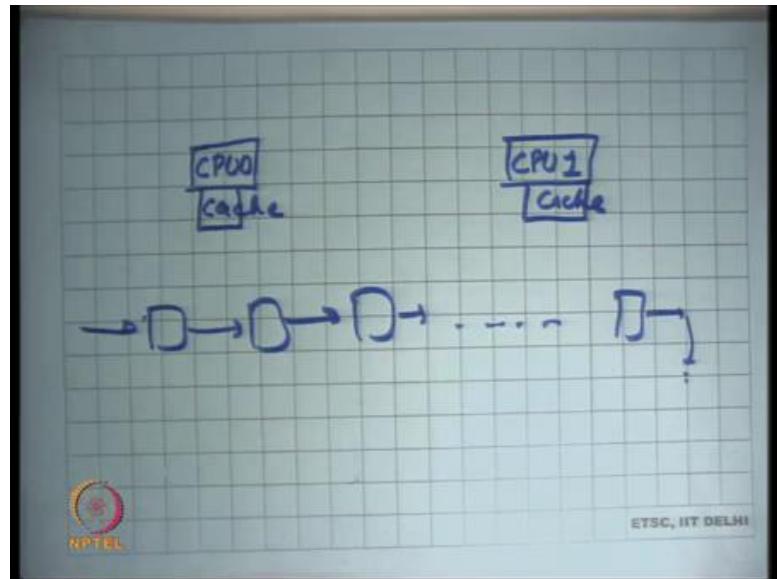
But it is quantum will be decreased, yes, all right ok. Also if you know if you have decided that a job is compute bound, so let us say it has moved down, so it has you know it is priority has been decreased, it is quantum has been increased, but you know let us say there is a process that was compute bound and then it suddenly became IO bound.

So, you know we are using the this assumption that past is equal to future, but past is not always equal to future. And so, the system should be able to adapt to such changes, and so there should be a way for the process to move up. So, we have discussed a way to for the process to move down in the priority hierarchy, there should always also be a way for the process to move up.

For example, you could say for thread blocks without exhausting its time quantum then you know you move it up with policy that you want to use right. And you can choose different policies. So, in general when you have chosen, we have chosen these policies you will favor recency over frequency. So, you will look at the most recent behavior to decide whether it is IO bound or compute bound.

You know you would not so just like a caching, caching placement algorithms in general you know you know recency is a much better metric to predict the future than frequency. So, we do not look at you know what is its behavior over the history of its execution, we just look at its behavior over the last time quantum or last five for time quantum something and that is good enough all right. So, that is you know a very simple notion of scheduling.

(Refer Slide Time: 32:00)



But actually scheduling is a relatively complex topic and you know subject of much research over so many years and still not completely mastered. So, let us see let us say these are two CPUs and there are lots of jobs that need to run. So, the question is how should one choose which job to run on which CPU and when? So, we have answered when to run which job, so you know there is some idea we have there.

We are going to do it in round robin order, we are going to have some priorities, and we are going to give higher priority to IO bound jobs and we are going to give lower priority to lower bound jobs. But does it matter which CPU I schedule them on? Well, actually it turns out it matters because CPUs have caches ok.

And so it is much more beneficial to schedule the same process on the same CPU repeatedly, because what will happen is the working set, the memory set of that particular process will in the respective cache will get warmed up with that working set and so you will have much fewer cache misses. And you know saving the number of cache misses is a significant optimization for any compute bound process ok, especially on modern systems.

So, then there is something that is implemented in operating systems today is what is called affinity scheduling. So, you run some sort of logic to basically try and ensure that the same process gets to run on the same CPU if it gets rescheduled right. If it is not a strict affinity, but it is a high probability affinity scheduling. Which means that a

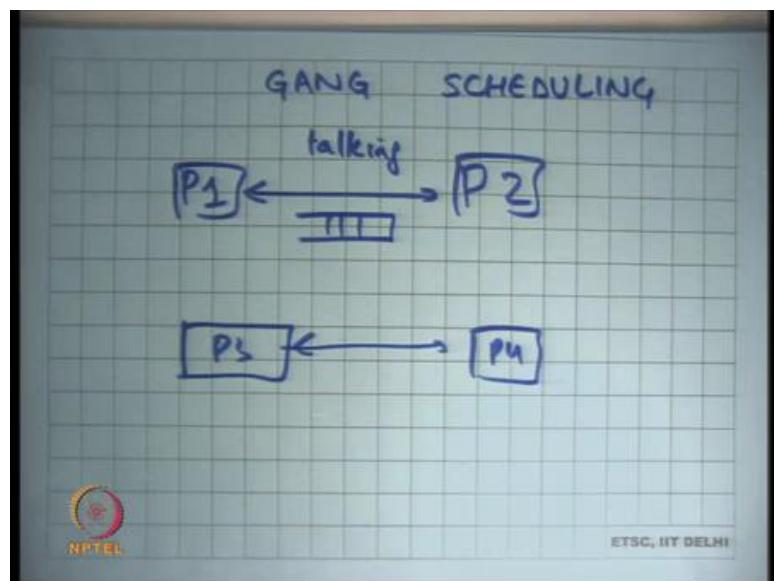
process has an affinity to a particular CPU, so whenever it gets it is scheduled, it will you know one CPU will be preferred over the other CPU and in scheduling it.

This can be implemented in multiple ways. You could have separate queues for each schedule each CPU, and you know so that these processes only get two schedule get two schedule on these CPUs if this CPU and these processes also get to schedule on all these CPUs.

Notice that this cache caching also has a significant bearing on the time quantum. Your time quantum should be large enough such that the process that gets to run has enough time to warm up its cache and then run on a warm cache. If most of the time on in that time quantum is just (Refer Time: 34:43) warming up the cache, and then you get scheduled out that is a very wasteful system right.

So, you should have enough time in your scheduling quantum to warm up the cache and then run on a warm cache for 90 percent of the time. So, let say 10 percent of time is taken to warm up the cache, and then 90 percent of the time is actually spent in running on a warm cache all right. So, you can have much better cache utilization if you are doing affinity scheduling.

(Refer Slide Time: 35:10)



There is also something called gang scheduling. So, if there are processes P1 and P2 that communicate with each other, let us say they have a producer consumer queue between

each other. Let us say one P1 is the network thread and P2 is the server thread, and they could talk to each other because they have a queue in the middle, and they will just keep exchanging information between each other right. So, there is a let us take an example, let us say there is a producer consumer queue or any other thing, so they are communicating talking to each other right.

And now there is P3 and P 4 that are also talking to each other. So, it will be a much better scheduling policy to basically say that P1 and P2 will get scheduled together. So, they will scheduled get scheduled as a gang, so that is called gang scheduling. So, you know P1 and P2 will get scheduled together. So, when they run, they are actually you know simultaneously, he produces he consume and consumes and so the buffer does not get full, there is no waiting and spinning and so it becomes very fast right.

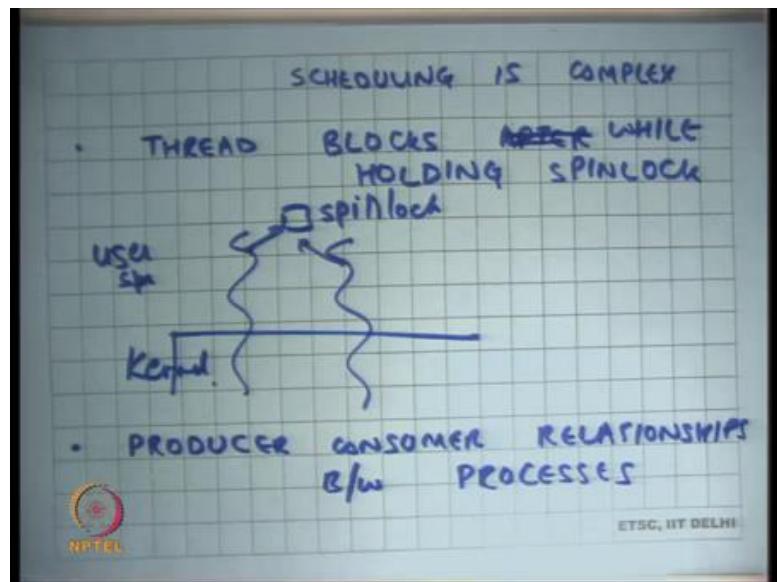
As opposed to let us say I did not schedule P1 and P2 together. So, P1 gets to run P1 fills up the producer queue, and then spins for some time just wasting or blocks which basically means there is extra overhead, then P2 gets to run, he empties the queue, then he is again spins to wait on an empty queue or blocks, and these are all wasteful operations as opposed to you know P1 and P2 running simultaneously.

So, it can actually significantly increase throughput for to communicating processes, so gang scheduling. So, the question is can the operating system always figure out that these are community communicating processes right. So, let us say; let us say there was there were two processes that were using a producer consumer queue that producer consumer queue was or there were two threads there were using a producer consumer queue when the producer consumer queue was completely implemented in user space.

So, the OS has absolutely no knowledge about this sharing behavior. It kicks you know they can they are they are some methods to figure out that there is some sharing, but you know those are two long winded to basically be really practical. If for example, you know these processes were communicating using a pipe, using an operating system pipe, then yes you know the operating system as full knowledge that P1 and P2 are communicating with each other.

He is writing to the pipe and he is reading those pipe and so the operating system can actually schedule them together. And there is of gang scheduling and get much better.

(Refer Slide Time: 37:37)



But in general, you know I am going to give you some examples to show that scheduling is actually complex all right. So, one example is thread blocks after holding spinlock or while holding instead of saying after that is a while holding spin lock. So, let us say there are you know there are 10 threads in the system, and they are all wanting to operate on some shared data and so they need to have a common lock right.

And so there is one thread that hold that takes a spin lock, and before it could actually leave the spin lock it either blocks which means that actually makes a you know it blocks voluntarily which means it makes an IO call let us say it makes a disk operation or there was a page fault right. So, page fault will also cause the thread to block right.

What will happen if there is a thread which is running and there is a page fault you basically just say that this thread is no longer ready you wait for the disk to bring the page into memory and you will let other processes run in the meanwhile that is what that is what we going to do. So, even if there is a page fault or if the process calls a system call which says disk read or any other IO call or if there was a preemption, there is a timer interrupt while you are holding the spin lock. All these cases are examples, where you are holding the spin lock and you have been de scheduled you know you are no longer running on the CPU.

And so, what will happen is all these other 9 threads will get to run, and they will just probably spin on the spin lock for the entire duration of the time quantum. So, they will

exhaust. So, spin lock is an example of synchronization which does not involve the operating system right, just like a producer consumer user left space producer consumer queue has absolutely no visibility for the operating system.

Spin lock is an example of a synchronization primitive that has no visibility for the operating system. So, the operating system has no idea that there is this something called spin lock spin lock is completely existing in the user space. Now, this thread actually takes a spin lock and gets de scheduled, all the other 9 threads just keep spinning on the spin lock, absolutely doing no useful work, but the operating system schedules them one after another and wasting a lot of CPU cycles all right.

What would have been a better way to do this? Well, if the operating firstly if the operating system knew that there is something called a spin lock that would have been the better thing to do. What does it mean for the programmer, so you know and there are other ways to do the to deal with these problems and they those ways to deal with this are have to do with operating system design?

So, in the Unix semantics as we have discussed you know this is possible and this is a very bad thing that can happen. And the only way the programmer can safeguard herself from you know not having this kind of a problem is to make sure that your critical if you are only protecting critical sections which are very small using spin locks all right.

Also, you make sure that within in those critical sections, you are not making any blocking calls. So, for example, you are not making any discrete or any other system calls that can take a long time. Also ensure try to make sure that you are not accessing too much memory, so that the probability of a page fault in that critical section is small all right.

So, these are some safeguards that you can take from to ensure that you know with Unix semantics of an operating system you basically do not have these kinds of problems where you know all threads wait for the spin lock. What could have been you know I am constantly using the word Unix semantics, what could have been the other semantics?

So, in the semantics that we have discussed so far, we are saying that the operating system has full power. It can take away the CPU from the operating system anytime it likes right. The other way to do it is you can tell the operating the process I want to take

the CPU back from you and the process has some idea of what are the resources it is holding. For example, it is holding spin lock and it can release those spin locks or you know finish up the it is critical section before it releases those spin locks and then saying here I am.

So, instead of a preemptive snatching of the resource from the process, it can be a more civilized you know requesting of the resource. You basically tell ask the process you know I need the resource back. So, you will say give me a second, I will just clean up a things, so that you know you have more best space throughput and then you know he gets at to you right.

Of course, you know this more civilized thing has of the problem that the process is not trusted. So, you know if you have asked him you know over the guarantee that he will actually obey you. So, in which case you know you also have some fallback mechanism. So, you have basically you know after you requested him, you wait for some time to you know, and if he does not behave as the good citizen you actually snatch it from him you know. So, these are ways to try to prevent this problem I am going to discuss some operating system models that actually do this kind of thing and it is very useful.

Student: You know (Refer Time: 42:35) spin lock is required by a CPU is not by a thread. So, as a how will it be possible that different different processes have been scheduled and all of them keep waiting for (Refer Time: 42:45).

So, question is you know spin lock is required by a CPU not by a thread. Well, that is not true, you know the spin lock that we discussed in XV6 was a per CPU spin lock, because you know we also disable interrupts, so that make meant there was no context switching. So, there was the spin lock was also asserted with the CPU, but in general spin locks have no need not be per CPU they can be per thread.

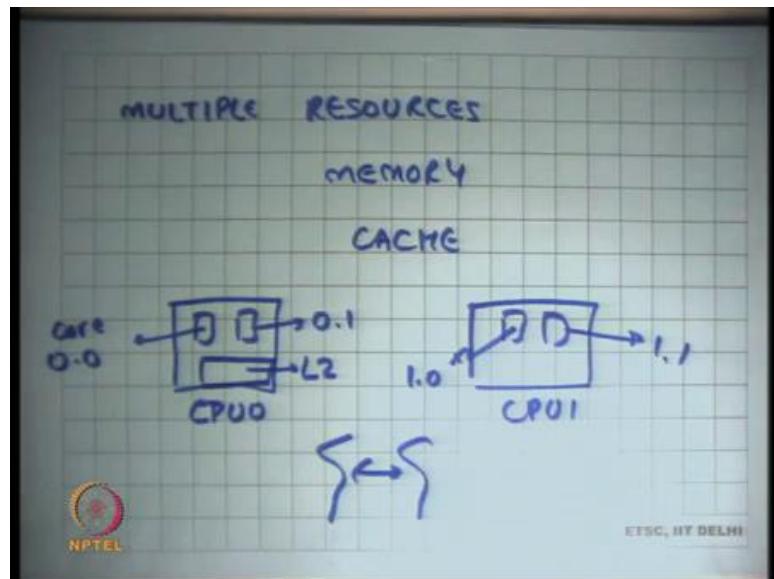
So, here is an example let us say you know there are threads, and these are this is kernel space, and this is user space, and they are basically having a spin lock here right. And so and this thread can call acquire on this and this thread can call acquire on this and the kernel has no idea that there is a spin lock that is going on. This acquire has nothing to do with clearing of interrupts or anything this. These are spin locks for threads all right.

Just like threads you know spin locks there can be producer consumer relationships. So, spin locks is just one example, but you know there can be any types of synchronization between processes right. For example, there is a producer thread and there is a consumer thread, and you know the producer thread has not actually filled up anything and it gets switched out. And the consumer thread gets to run and lots of the consumer threads.

So, all the consumer threads will just let us say the way you are implemented the producer consumer queue is completely at the user level, and these threads spin if you consumer thread spins if it finds the queue to be empty, and the producer threads spins if it finds a queue to be full. And so, the operating system or the scheduler has no idea that there is a producer consumer relationship and it will keep scheduling the consumer threads only for them to spin and do really no useful work all right.

It would have been much better if there was a way for the operating system to know this or better the operating system would tell and indicate its intentions to the operator to the process that I want to take the CPU away from you, or I want to schedule you, and get it is feedback on you know how much time you know when I should do that, what is the right way time to do that all right.

(Refer Slide Time: 44:55)



Also, there are multiple resources. So, I am talking about CPU, but they are actually multiple resources like memory all right. So, if I know if I have a high priority process let us say vi and gcc example, and I am giving lot of CPU I am giving vi higher

priority over gcc. But that is of no use if I am not giving vi also the amount of memory that it needs right. If the higher priority process is not getting the same amount of memory that it needs, then the first thing it will do when it gets scheduled is block by taking a page fault.

So, priority does not need to be obeyed only on the one resource with this CPU, it also needs to be obeyed or there some needs to be some reflection of it on the other resources, in this case the physical memory right. So, for example, if there is a higher priority process perhaps it you know it is working set should be in memory as opposed to the lower priority process.

Of course, once again there is an issue of you know one process should not get be able to run away with all the memory. And so there has to be some the some safeguards in scheduling the memory. So, there is not just scheduling. So, they are scheduling in time of CPU, but there is also scheduling in space of the memory and these to need to work in tandem with each other that makes things a little difficult.

And we have also talked about cache scheduling. So, cache is basically caches in even scarce resource then memory. So, you know for best performance you would want that the cache is properly scheduled, affinity scheduling being one example of a good way of scheduling a cache.

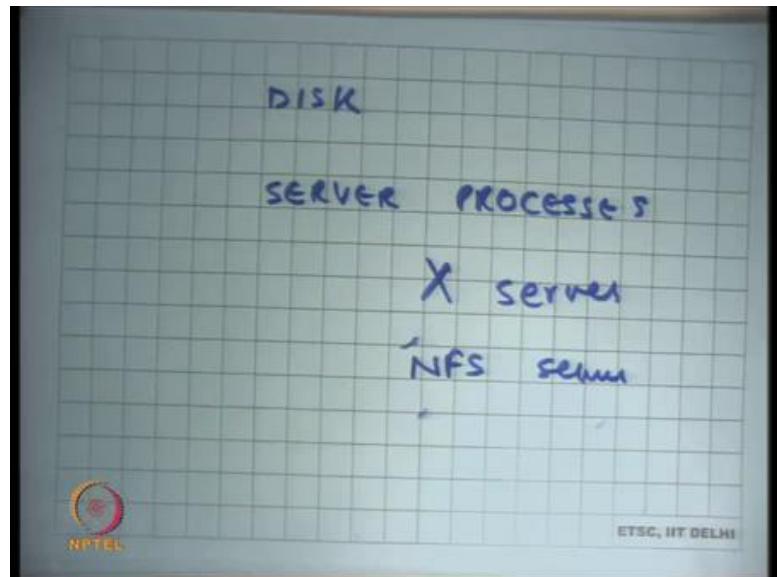
And there you know similarly if there is a producer consumer relationship for example, let us say you know you have CPUs. So, these days you get multi core CPUs. So, let us say there are two CPUs which have two cores each. So, this is CPU 0, and this is CPU 1, and this is core 0 dot 1 and this is core 0 dot 1 this is 1 dot 0, and 1 dot 1.

If there are two processes that have a producer consumer relationship between them then it is better to schedule both of them on the same CPU together right. So, that way you are efficiently using the cache, because each CPU here will have L 2 cache that they can share between themselves as opposed to going to the main memory to implement to do this producer consumer relationship right.

So, there is a very complex thing here that you know you first need to figure out that who are having the producer consumer relationships, then you to figure out you know what

are the parts of the CPU which CPUs have shared caches, and then schedule those processes in tandem in a gang scheduling way to do that all right.

(Refer Slide Time: 47:30)



Then you know similarly this disk. So, for example, you know there are page faults from multiple processes, there is a page fault from the vi process and there is a page fault from the gcc process the disk schedule has no idea which has which requires has higher priority and which should get scheduled first. And you know either I you know either I do not care about it in which case I am not actually going strict priorities.

There are other parts of the system that have no idea about your priority system or I you know flow this information down in which case I can severely impact my throughput right. So, if I do; if I do strict priority scheduling at the disk level, then you know it is I cannot and can no longer follow the elevators algorithm and maximize my throughput.

Also, there are server processes like for example, the X servers right. So, X server runs as a separate process and all the other processes talk to the X server to you know make their requirements for displaying or rendering their screens. And so, the X server has no notion of priority or you know it is very you know it will require a more interfaces to ensure that the X server also has some notion of priority.

But ideally if you know one process a higher priority over the other process, his request should be given more priority over the other processor priority. So, I do not care about

gcu screen so much, but I definitely care about vi screen for example and so you know how is this kind of a thing done. From a schedulers perspective he is blind to you know what is up he just sees that the X server wants to access the display device, but the X server internally has no idea that you know which required which process has higher priority or not.

One could build it in, but you know that makes life complex there are multiple things that are happening here. In another example is an NFS server right. So, NFS server is a network file system, you could have a separate server and the multiple processes that are accessing this NFS server. But the NFS server needs to be either made aware of the priorities or you are not actually following strict priorities ok, so good so that is it for scheduling.

I am going to discuss one very interesting example of non-linear effects of scheduling. So, scheduling in general is a problem if the resources are relatively scarce right. So, if you have lots of resources let us say CPU is plentiful and you do not care about it, there is no problem right, scheduling is not an issue, but the moment resources become scarce it becomes an issue.

If you are running a web server that is handling a 100 or 1000 requests per day the performance of a scheduler will actually not matter at all. But if you are running a web server that you want to operate at peak capacity let us say hundreds of thousands of requests per second, then you know the way you schedule your request and the way you schedule your resources becomes very important right.

You are you know you have you are bottlenecked by your caches, you are bottlenecked by your disk throughput, you are bottlenecked by the CPU and so the an efficient scheduling and inefficient scheduling there can be a large gap in performance between the two.

In fact, the gap can be so large that at high throughput I am going to show you know we are going to discuss a an example of an where a poor scheduling can actually lead to zero throughput at very high loads ok. So, there is a non-linear effect, it is not necessary whereas, a good scheduler can actually you know saturate very gracefully in presence of high load right.

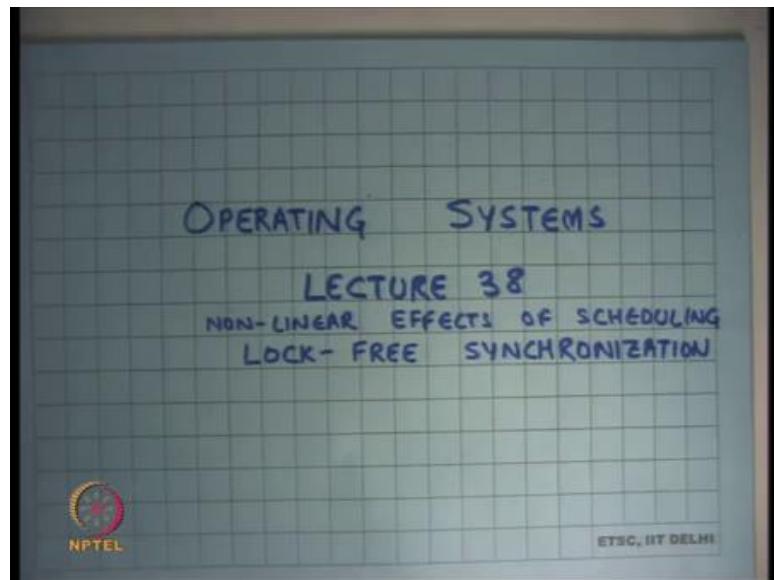
So, overall, I mean scheduling was a very important topic in the early days of computers 1970s, where this would be large, shared computers that are very low computing power and lots of people sharing them. Over the years the computers became more and more powerful and more and more distributed. We had personal computers and each and so fewer users and more power and so scheduling was less of a problem is less of a problem on a desktop or laptop PCs.

But you know we are moving back to you know let us say what is called cloud computing, where you have a centralized computer or centralized data center and being shared by lots of people. And once again to maximize the utilization of your resources, so and so once again you know to operate your centralized system which is shared by lots of different processes scheduling becomes again important in the modern world as well good. Let us continue the discussion next time.

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

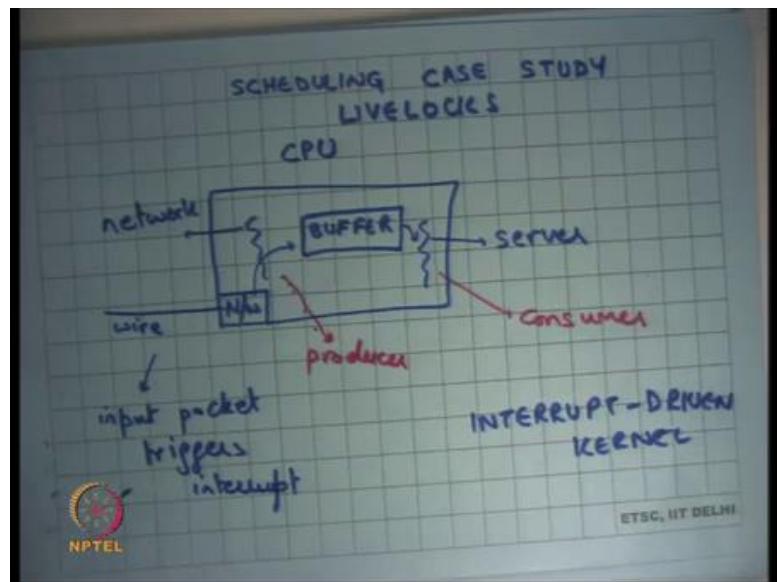
Lecture - 38
Lock-free multiprocessor coordination,
Read-Copy-Update

(Refer Slide Time: 00:24)



Welcome to Operating Systems Lecture 38. So, I am going to first discuss continue our discussion on Scheduling, I am going to discuss an interesting case study where we will see that badge scheduling can have very non-linear effects in terms of performance. And, then we are going to talk more about Lock free synchronization after that right.

(Refer Slide Time: 00:44)



So, we have been looking at scheduling and what are the different scheduling policies and we talked about priority scheduling and we said that typical general-purpose systems use some sort of priority scheduling. Where they prioritize IO bound jobs or interactive jobs over compute bound jobs or long running batch jobs and that ensures that there is a high throughput and also very high very low response times right. But these priorities are not strict in nature in the sense that priorities keep changing.

So, if a job has not been received attention of the CPU for a long time, then gradually its priority will increase and so eventually its priority will become maximum and there is no starvation problem. So, in general purpose systems you avoid starvation problems of this kind, but in other systems like real time systems let us say you are running an operating system in a car, there you will want to implement strict priorities.

So, some processes are strictly higher priority than other processes. For example, safety related processes will have higher priority than you know other processes that are not so critical ok. But I am going to talk about one case study which will show which will you know give you a nice insight into the non-linear effects of scheduling.

So, if you do a bad scheduling then actually your system can behave rare very poorly under high load. So, we also said that scheduling is a problem when the resources are saturated in their you know; so, if the number of resources are short for the amount of or the rate of requests that you are getting that that point scheduling becomes really

important. If the number of resources is plentiful then scheduling is less of a problem of course alright.

So, let us take this example let us say this box is a computer or you know CPU and let us say this it is connected to the network and this is a network card let us say network device and you know packets are coming in. So, it is a let us say it is a server and their packets coming in from the wire and this let us say it is a web server or you know some kind of a network server and it is receiving packets and then it is sending a reply. So, it is receiving a lot of requests and then sending replies to those requests.

Now, you know one way or one of the typical ways you would implement such a server is that you will use interrupt ways mechanisms to handle incoming packets. So, as soon as you receive a packet you know you will configure your network device or a network card to say interrupt me whenever you receive a packet right. Why do you say that? You know so that as we know that two ways to handle IO, one is interrupt based IO and the other is polling based IO.

So, there are two options here one is I could say I could configure the network card to say every time you receive a packet interrupt me right and interrupt is looks like interrupt is nothing but a you know very high priority processor. So, in interrupt basically preempts all other lower priority processes, assuming there was no other you are not already inside an interrupt handler; whatever was running will get preempted and you will get to run the interrupt handler. And the rationale behind that is that if a packet that has come in you want to serve it immediately right.

So, there is a request that has come in you want to minimize the response time of the request. So, the reason you would want to have an interrupt-based system is to minimize the response time. The other approach you could have taken was not configured the network card for interrupts and rather said I am going to check the network card every few milliseconds or you know some granularity. So, let us say I checked the network card every 100 milliseconds.

So, in which case you know the network card needs to have a buffer and packets are coming in then the buffer you know it does not need to get buffered in that buffer on the card. So, it has nothing to do with the CPU there has to be a buffer on the device itself

and then the CPU can check the network card for packets. What is the problem here is that the response times are higher than what they can be?

If a packet comes on average it will have to wait on the buffer for 50 milliseconds, you know if you are if I am pulling every 100 milliseconds then my response time average response time will at least be 50 milliseconds which is very large. You know I could have done much better if I was using interrupts, then I could have given much, much better response times on the order of microseconds let us say alright.

But so let us say I have an interrupt driven kernel. So, I am going to say it is an interrupted driven kernel right. So, which means that each time I get a packet I get an interrupt and the packet gets copied from the network card to the Buffer, to some buffer that I am maintaining in the main memory right. And, then there is another process server which is which is consuming from this buffer and doing some processing on it. Let us say you know it is formulating a reply and sending a reply back on the network card or some other network card doing something with the packet right.

So, with an interrupt driven kernel you know I modelled this as a producer consumer, where the producer is the network card and the consumer are a server thread. And, an interrupt driven kernel the producer is basically always a higher priority process than the consumer. So, if the producer has something to do for which means it has some packet to produce inside the buffer, it will get to run, and it will preempt the consumer every time right ok.

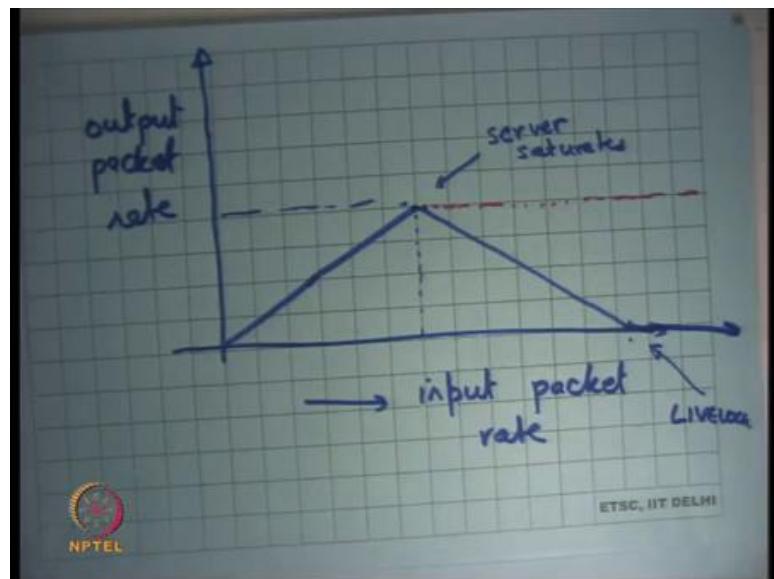
So, this works this has very low response times because, you know you will basically immediately produce and consume and things like that. But, let us see what happens if there is a very high load on the network card. So, let us say now I have a very high-speed network link and my CPU cannot process links the packets at that rate right. So for example, you know I am receiving packets at the rate of you know on I am using a 10 gigabyte per 10 Gbps link and I am getting lots of packets and the processing I need to do on the CPU is cannot sustain that rate of packets ok.

So, what is going to happen in this case? The packets will start getting dropped inside this buffer. So, this buffer will fill up and the packets will get start getting dropped at this buffer alright.

What will also happen is that because you are interrupting on each packet and the interrupt rate is so high, because you are getting packets at a very high rate the server thread will never get to run. So, the consumer will never get to run right. So, that is only the producer that is getting to run, and the consumer is never getting to run, which effectively means that the CPU is actually doing no useful work.

All it is doing is getting the packet from the network card to your buffer and there is no follow up on the buffer and eventually for the packet to get dropped. So, you know this act of actually bringing the packet from the network card to the buffer is completely wasteful in this case right. Because, that packet is destined to get dropped eventually right or in fact you know if the buffer is full then it will just get dropped right there right.

(Refer Slide Time: 07:57)



So, this is an example of a very bad scheduling right. So, what will happen is if I was to draw a curve between let us say if I wants to draw a curve between input packet rate and let us say output packet rate. Let us say my server was sending a reply for every packet right. So, let us say there was an input packet rate and output packet rate, this curve will be pretty much linear for low rates. But, at some point when the input packet rate is so high that the server actually gets saturated, you will basically start seeing a drop in the throughput till it you actually reach 0 right.

So, this is the point where the server saturates and the reason you are basically seeing a drop is basically because, after the input packet rate becomes higher than a certain limit

then the producer gets to run more often and the consumer gets to run less often. And, so because the consumer is getting to run less often you know the net throughput is dependent on the consumer right which is the bottleneck. And, so the higher the input packet rate the less often the consumer gets to run till the consumer does not get to run at all no.

So, the input packet rate becomes so high that the consumer does not get to run at all and at admit point you basically have a net output rate of 0 right. So, this is the point at this point we say it is a live lock system right, we call a system live lock if the system is doing a lot of work which making look progress right. So, here is this is definitely a case of a situation where the system is doing a lot of work it is continuously the CPU utilization is 100 percent, yet the output net useful output of the system is actually 0. So, what is the problem how could I fix this problem?

Student: If the rate is very high, we can switch to polling.

If the rate is very high, then I could have switched to polling and so you know what is the best good answer and so what is the best I can expect. So, let us go in the reverse direction. So, I do not like this fact that is the input packet rate is. So, high then my throughput becomes 0. What is what would have been the ideal system?

Student: Freezing and then saturating.

Right, so the ideal system would have been that at this point the CPU got saturated, clearly I cannot serve anything which is more than that after all the server cannot do anything more than that, after this I should have had a curve like this right. So, after that you know all the packets that are coming no extra rate is getting discarded. But the rate that I can support at least that much is getting served right, that is the ideal thing you could have had in your system right and why am I not being able to achieve this ideal behavior.

Because, I am doing bad scheduling right because I am doing bad scheduling because I am prioritizing the producer over the consumer irrespective of anything else right and I did that because I wanted very low response times right. So, here is a very nice example where because of trying to minimize the response times I actually get a very bad throughput in my system alright. So, and so what is happening is that only the producer

of the queue gets to run, and the consumer never gets to run and so you basically degrading your throughput. So, how would you reach this point from this point, you would want to reprioritize things you would have want to say that ok.

You know I can see that there is some problem you know I want to give equal priorities to my consumer and producer at some point. It should not so happened that producers getting to run too much, and the consumer is not getting to run at all right or but consumers getting to run less than the producer, that to me is a warning sign right. That is if the consumer is getting to run less than the producer or in other words if the consumption rate is less than the production rate then that says it looks like a warning sign and what I should probably do is throttle my producer. So, that you know the net throughput remains nice, otherwise if the producer just running is just doing wasteful work right.

So, the idea is this thing that could have happened is that instead of the packets getting dropped at this buffer, the packet should have been dropped at the network card itself right. In that case the CPU will not have done any wasted wasteful work right what is happening is that the work that is required to bring the packet from the network device to the buffer is completely wasteful in the case of we know when you reach the live log stage.

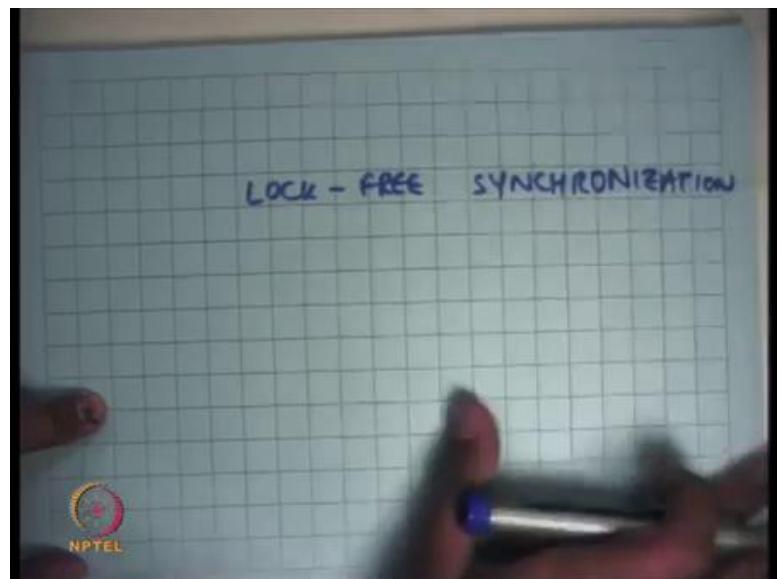
But, if you somehow figured out that no you know I do not need to do this wasteful work, then you could then what would happen is eventually the packets would get dropped here right. And, one way to think about this is that you know if you notice that your buffer is becoming higher you know if the buffer lengths average buffer lengths are becoming higher than a certain threshold. Then you throttle your producer throttling your producer basically means that you switch from interrupt mode to polling mode and you have to choose your polling frequency right.

The nice thing about switching to polling mode is that you choose you can choose your polling frequency and the choice of polling frequency determines the rate of execution of the producer or the priority of the producer right or the proportion of the producer. Eventually I want that the producer and consumer rates should be similar and so you know that is how I would ensure that kind of thing right.

So, the solution to this says that you know which is which is let us say implemented in mainstream kernels and so this was a problem in the Linux kernel. You know let us say 15 years back it was discovered and so the solution that was implemented after that was that you use interrupts. If the rates are low and then you at some point you switch to polling when the rates are high, and you choose your polling frequencies depending on your average buffer lengths and the time it takes to basically.

So, the time the producer and consumer processes are getting to run right. So, here is an example where there is a tension between response time and throughput, and you want to have both of them dealt well. So, in general you know the thumb rule is that interrupts are better at low at low rates and polling is better at higher rates right ok.

(Refer Slide Time: 14:22)



With this I will switch to my next topic which is Lock Free Synchronization. So, we have discussed some lock free synchronization before, and I am going to discuss more of that today alright.

(Refer Slide Time: 14:35)

```
SEARCHABLE STACK
struct element {
    int Key;
    int value;
    struct element *next;
};

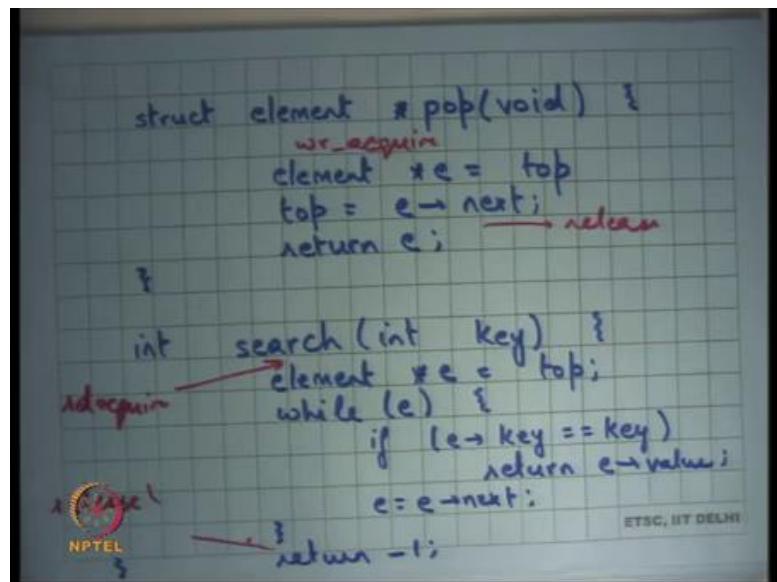
struct element *top; //global
rwlock rwlock;

void push(element *e) {
    wr_acquire(2rwlock)
    e->next = top;
    top = e->next;
    release( . - )
}
```

So, let us and let me first introduce to you a small example, let us say I wanted to implement a data structure called a Searchable Stack. The searchable stack has elements of this type an element is a key and a value, and a next pointer and the stack is represented by a variable called top right.

So, you can only look from you can only start at the top, you push to the element by just incrementing talk basically you wanted to push an element e you just say e.next is equal to top, top is equal to e.next (Refer Time: 15:10) that is pushing an element onto the stack.

(Refer Slide Time: 15:13)



And, you top an element by just decrementing top you say e is equal to top, top is equal to e.next and e right. Of course, I am gliding over some details like if top is not equal to null and etcetera you can do that. But there is a third operation which is a search which just goes through the stack and searches for a particular key.

So, you want to search for a key you just iterate over the stack while e fe.key = then return you got value otherwise just do e.next, if you find it very good if you do not find it will turn this right. So, there is a searchable stack the three the two operations push and pop which are write operations read write operations. And, there is one operation which is a read only operation called search you know this search is the search as stack clearly in a if this code is executed concurrently, if multiple threads are sharing the stack searchable stack then there is a problem right.

I am going to discuss that we all know there are concurrency problems, in this course there is a concurrency problem if somebody is searching and somebody is pushing at the same time and also there is concurrency problem. If somebody is pushing the two threads are pushing at the same time or pushing and popping etcetera. So, what are some ways to solve it well one way to solve it is usually use locks right. So, you basically say I have a lock for the entire data structure. So, and basically put a lock acquire and release around this you put a lock acquire and release around this and you put a lock and acquire and

release around this, that is a coarse-grained lock right. And that will severely impact your performance it will not scale at all.

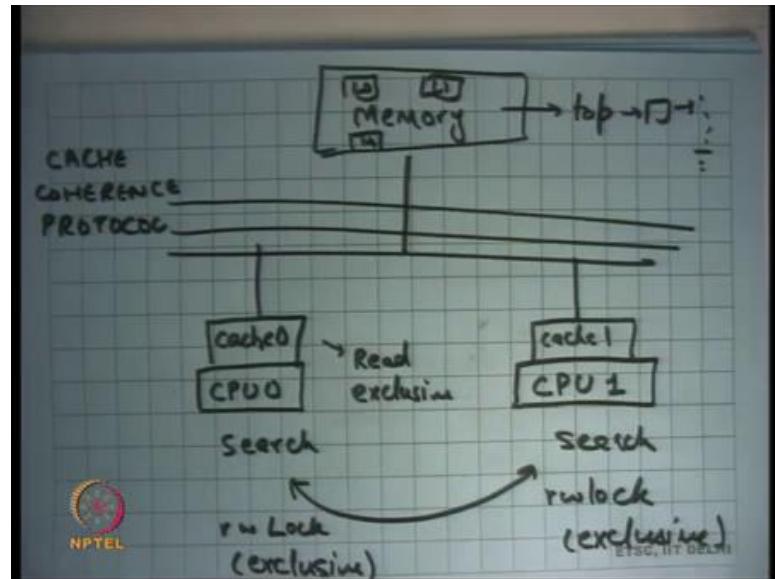
So, if you have multiple threads then you will only have single threaded performance, let us say this code was the only code that is running then you will you basically have no scalability. What is the other thing you can do let us say I tell you that most of the times you are going to execute search and push and pop are relatively rare operation?

So, you know just as an example 99 percent of times it is search that gets called and only you know one percent of times it is either push or pop that it is called. So, it is mainly search that is getting called and search is read only operation. So, what will you do reader writer lock right? So, simple I will just use the reader writer lock, I will say you know I will have I will have.

So, I will declare a reader writer lock let us say rwlock and then I will have write acquire rwlock and release and I will have similarly I will have write acquire here and I will say release and here I will say read acquire right and I will say release here. So, why is reader writer lock better than your normal lock because the semantics are that multiple the lock can be acquired in read mode multiple times simultaneously.

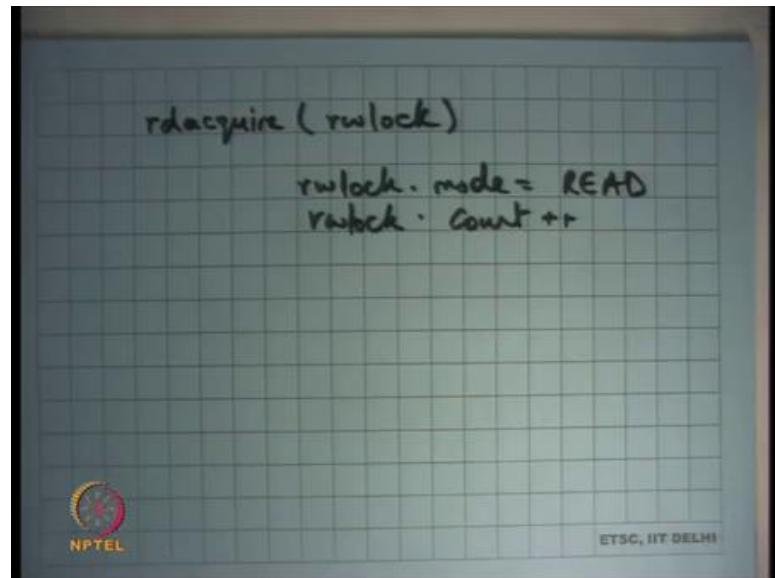
But the lock cannot be acquired in simultaneously in write and read mode or simultaneously in write and write mode right. So, there can be ninety nine percent of times you are calling search and all these threads can execute simultaneously. But is it really better than is it really that good? So, let us look at what is really happening at the CPU level or at the hardware level ok.

(Refer Slide Time: 18:43)



So, let us say you know I have CPU 0 and CPU 1 and let us say I have a shared bus and I have caches here. So, have cache is 0 and I have cache 1 right and we have connected to the bus in this memory ok. Before I discuss this let us let us also revise how reader writer locks are implemented.

(Refer Slide Time: 19:20)



So, you know if I want to implement a reader writer lock, I will say you know how will I implement read acquire you know I will say I will set some flag and say rwlock.mode is equal to read and count++ right. Something of this sort and you know I must make sure

that it is not already held in write mode right. So, I am going to and so this I am going to have to write to this lock variables I am going to I have to maintain a state in this lock variable that it is read it is held in read, read mode. And I will have to also write a count variable it says you know I have I have basically acquired it in read mode.

So, that other readers know that, or other readers and writers know that acquired. So, what has happened is ah? So, what I am going to show you is that this code which was actually completely read only code has now become a read write code right. So, here I am only doing read operations on all the shared values. What does my shared value, the shared value is my stack and all I am doing is I am just doing reads on the stack right. The only rights I am doing is to my local variables which can be you know my local stack or my local registers, whatever but that is not that is not shared so I do not care about that.

So, the only things that I am the only operations that I am making on the shared variables are reads, but the moment I put a read acquire and a write acquire these operations involve a right to a shared variable all right. And that is a bad thing and so why is it that a bad thing? Because if I look at the hardware level, if there is shared data let us say there is there is a stack living here let us say this you know top this is the searchable stack living here and so on. Then if I was if both of these were calling search, then very likely this stack would get cached in the local caches of both these CPU.

And so, these CPU can just execute search at cache speeds, they do not need to go to the bus to be able to read any memory locations right. Assuming that all local variables are already in the cache or in the registers, the global variables if the access of the global variables are read only accesses then you know modern hardware allows you to cache both of them simultaneously in read mode only right. So, before so just some background: hardware implements what is called a cache coherence protocol, how many of you have studied a cache coherence protocol not really alright.

So, a cache coherence protocol basically says, you know it allows so for every memory location m memory location M can be cached here. So, it basically maintains coherence of data which means that a memory location M can be cached here in read mode or exclusive mode right. If another cache accesses it also then if the cache memory location is held in read mode, then this can also cache it. And so, the same location can be held

can be cached read mode and multiple caches simultaneously. But a location can only be cached in exclusive mode in one cache at a time right this is clear.

So, every location can either be cached in read mode or in exclusive mode, if you know a location can be cached in read mode in multiple caches simultaneously known to increase performance and because it is in read mode nobody can modify it. If somebody starts to modify it what happens is all the other read mode cache entries get invalidated and this the one that modified it now holds it in the exclusive mode alright. So, it says that and this mechanism to implement this kind of protocol is implemented in hardware and it is called the cache coherence protocol right.

So, if the CPU are accessing the list in read only mode, what will happen is that this list will get cached in the local caches of the CPU in read mode eventually. And what will happen is that all the search operations will only access the local caches and never have to go to the bus right. Recall that you know cache speeds are on the order of few nanoseconds in one or two nanoseconds. On the other hand, a bus transaction involves depending on you know whether you actually go to the bus or whether this scores are within a single chip, it can range from tens of nanoseconds to hundreds of nanoseconds.

So, it is on that order, so it can be 10 to 100x slowdown from just cache accesses right. So, if you just if you only have cache it is you versus if you do not if you only have cache misses you can have a 10 to 100x performance difference in the thing. So, earlier you all the things were getting cached in the local caches and so both the CPU could have executed search at full cache speeds.

But now because you have a read acquire and write acquire there is another shared variable here which is the lock right. And because the lock is being accessed in read write mode it is been modified, what will happen is that the lock will get cached here will need to get cached here in exclusive mode. Then this if you access try to modify it and so it will get need to get invalidated here and now it will get cache here in exclusive mode and.

So, the lock will keep bouncing the read write lock will keep bouncing between the two caches right and because it needs to be cached in exclusive mode, because it is a write access to these variables right. So, each time you access if you each time you write to the

read write lock you actually need to invalidate the value in the other cache if it exists to get it to get the latest value right.

And so, what will happen is that this read write lock will keep bouncing between the two CPU and each time it bounces it involves the bus transaction and that gives you a lot of slowdowns alright. So, what can happen is that this code will actually becomes it is possible that this code actually becomes slower with read write locks then faster right.

So, for example, if you have two CPU and if I tell ask you to implement the fastest possible you know fastest possible a solution to my program which involves accessing the read write lock. You may say let me use a read write lock, but if you use a read write lock because of the cache line bouncing the total performance of two CPU may actually be lower than the performance of one CPU.

Because if you only ran the program with one CPU; it would have been a cache line access cache access. But, if you access a two CPU if you paralyze the program with read write locks even though there is full concurrency, because of cache line bouncing you have actually slow downed the program by a factor right.

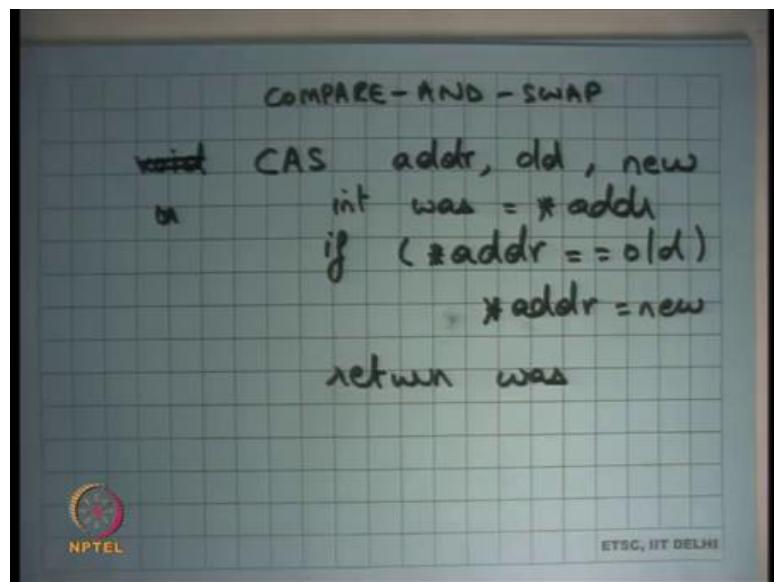
So, the advantage of actually having more concurrency is gone in some sense right. In other words, you even though you know the CPU are called are executed concurrently the bus becomes a serialization bottleneck in some sense all right. So, what was the problem, the problem was that I had a read only function and by using read write locks I converted it into a read write function and that in that was that made it less scalable right.

This problem I have demonstrated this problem on two CPU, but this problem becomes even worse actually much worse if you go to larger number of CPU. For example, you know today you can get a machine which has eighty CPU in it you know a proper and you know a full desktop machine desktop processor like the Intel for example.

And so you can you can get a machine with an eighty CPU machine eighty CPU eighty CPU on it and then you have a cache coherence protocol going on between the eighty CPU. And, now this lock is bouncing between the eighty caches and so that can become a huge performance bottleneck all right. So, firstly it clearly shows that you know reasoning about performance and concurrent programs is extremely hard.

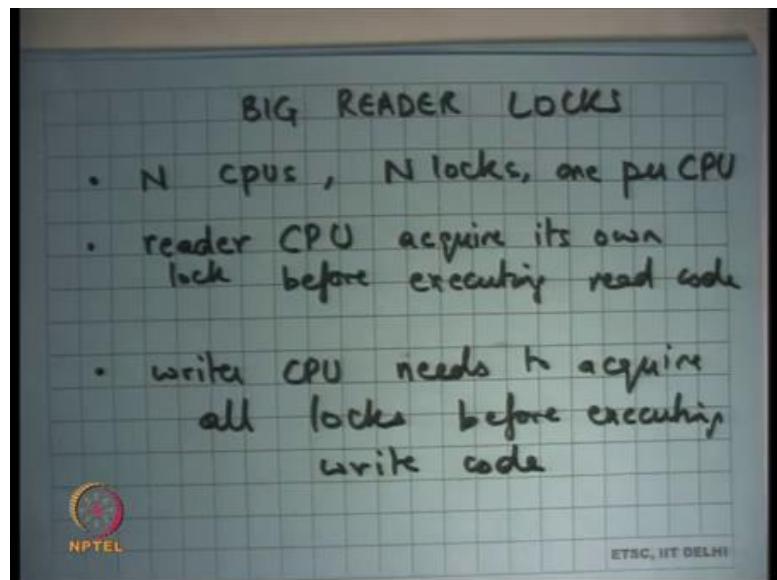
But what could have what could I have done, basically it is a bad idea to convert a read only code to a read write code right. It would be nice if I could not, I would not have I could have done this synchronization without having to do this right. Why do I need to have this read write operation inside my read only function, because I need to synchronize it with other write operations right that is the only problem I have? So, what could I have done right let us say I used a lock free primitive like compare and swap?

(Refer Slide Time: 28:03)



So, we have seen compare and swap before and I said that I am going to use compare and swap to implement my writes and because my compare and swap is going to be atomic my reads do not need to take a lock let us see if that if this works alright. So, what I am going to do is I am going to implement ok. Before I talk about compare and swap let me talk about one alternate solution.

(Refer Slide Time: 28:33)



So, here is one alternate solution what are called br locks Big Reader Locks alright. So, big reader locks have this let us say there are N CPU, or you know you can generalize it thread, but I am going to consider N CPU. Then you have you know you have N locks one per CPU, reader CPU I know I am just using a term thread and CPU interchangeably. Reader CPU acquires its own lock and before executing read code and writer CPU needs to acquire all locks before executing write code ok.

So, idea is that the, so you have a lock which is a large you know which is a structure which has now instead of one bit or two bits it has N bits. And the idea is that every CPU will acquire its own lock it is own bit and the other CPU, so and the writer CPU will need to require all the bits right. So, why is this better than the reader writer lock that we just discussed let us see. What will happen is that the let us say the CPU 0 and CPU 1 so you have two locks CPU 0 will have its own locks.

So, there are two locks L0 and L1 CPU 0 will make a write operation, but it will make a write operation always to the same location L0 right. And so, what will happen is L0 will get cached in cache 0 and L1 will get cached in L1 both in exclusive mode. But there will never be any bouncing between the caches. So, L0 will never need to go to L1 and L1 will never go to L0 and so they will both live on caches and yet you will basically ensure that it is correct right because you know.

So, readers can execute concurrently because you know one CPU has acquired L0 and other CPU acquired L1. But a reader and the writer cannot execute concurrently, because the writer will acquire all locks. So, you know it will writer will need to acquire all locks. So, what you have done is you have slowed down the writer, but you have really made the reader faster right. And that is the case we are optimizing for I am saying that the reader is executing ninety nine percent of times and writer I do not care about the writer performance at all ok. So, this works what are some problems.

Student: Sir the writer can start.

The writer can start well, not really. I mean it just depends on you know. So, it just depends on your lock acquisition algorithm. So, you can just say you know I am going to give locks on a first come first serve order. So, just like a reader writer lock you can say that you know, if some if a writer makes a request for a lock li then you know it will eventually get the lock within a finite amount of time within a bounded amount of time let us say.

It is the writer cannot stop, but the writer actually has to make a global operation if you are sixty you know if you have eighty CPU and tasks to do a lot of a lot of work to get there. The more important thing is that these caching is not done at bit granularity right caching is done at cache line granularity right. So, you basically just so just like pages, pages are a unit between disk and my main memory, the unit between a main memory and the cache is a cache line right.

You cannot just say I want to cache this byte you have to say I have to want to cache this line cache line that contains this byte right. So, it is a full cache line that you have to get into the cache right. So, what does that mean for our big reader lock can li means sharing the same cache line no right. So, all these different reader locks n locks need to be on different cache lines in. So, basically you will need to make sure that the variable li is occupying one full cache line right.

So, typical cache lines let us say will be 64 bytes on modern hardware you need 64 bytes for every CPU. So, if you have 80 CPU and you have 64 bytes into 80, so the lock structure actually becomes pretty large and that is it right. Moreover, you know you have to you cannot write generic code it literally depends on what the cache line is on the

architecture. So, you basically need to check what the cache line is and compile your code for that particular for that particular chip right.

Different chips for the same architecture can have different cache line sizes and you need to optimize based on that. So, these are all you know difficult things to do. I mean imagine if I was to use big reader locks and the Linux kernel then to sort of worry about which chip is this kernel going to run on to basically decide what say structure of my big reader lock should be so.

So, this is not an option, but, but let us look at a better option you know which does not have these problems and I am going to try to use compare and swap to basically implement lock free synchronization and in this case example in a better way right. So, what I am going to do is. Firstly, why do I want to void locks, let us just also review why we want to void locks.

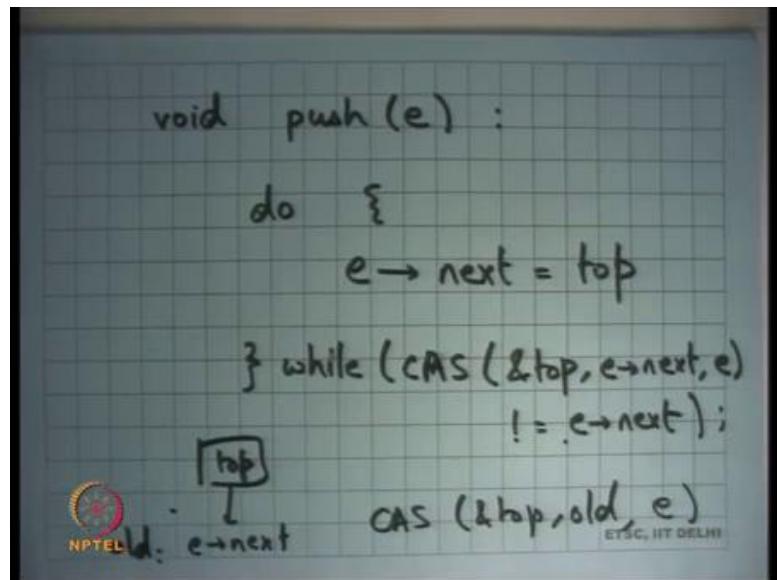
Firstly, I have already discussed locks performance problems you know the cache line bouncing, even for read only code even if I read use reader writer locks. There is complexity involved in locking you have to worry about you know fine grained locks for this coarse-grained locks etcetera, you have to worry about deadlocks, and you have to worry about priority inversion ok.

So, we have also studied scheduling and there is a problem with locks, locks are just resources and people need to debate on resources. So, you know anytime you have something like that then this priority inversion that is a problem ok. So, you know let us look at the let us look at the other way of doing locking synchronization is compare and swap, you know which is which is sort of a transactional way of doing things. We will basically let me implement to push and pop using compare and swap, so let us just revise for what is compare and swap.

Compare and swap is of an instruction let us say let us say CAS that takes three arguments one is an address; another is the old value and the third is the new value. And the semantics are if the contents of address are still equal to the old value then replace them with the new value right. And, you do this in an atomic way also this instruction returns which means it overwrites one of these registers with the value that was read.

So, let us say I added a value which was you know let us say I said was is equal to star adder and then I return was. So, I read this location and I compare, and I returned that firstly always. But I also compare it with the old value and if it is equal then I replace that location with the new value and this entire equation is atomic right that is a compare and swap you compare, and you swap. If the comparisons are successful, if the comparison is not successful then you do not fall right.

(Refer Slide Time: 36:43)



So, that is the compare and swap and let us see how we use it for something like push. So, I say void push element e I am going to say e.next is equal to top right and then I wanted to do top is equal to e.next. But I wanted to do that top is equal to e.next only if top has not changed in the two between the two are instructions right. So, what I am going to do is I am going to say do while compare and swap top address old value which is what you read e.next and new value e is not equal to e.next right.

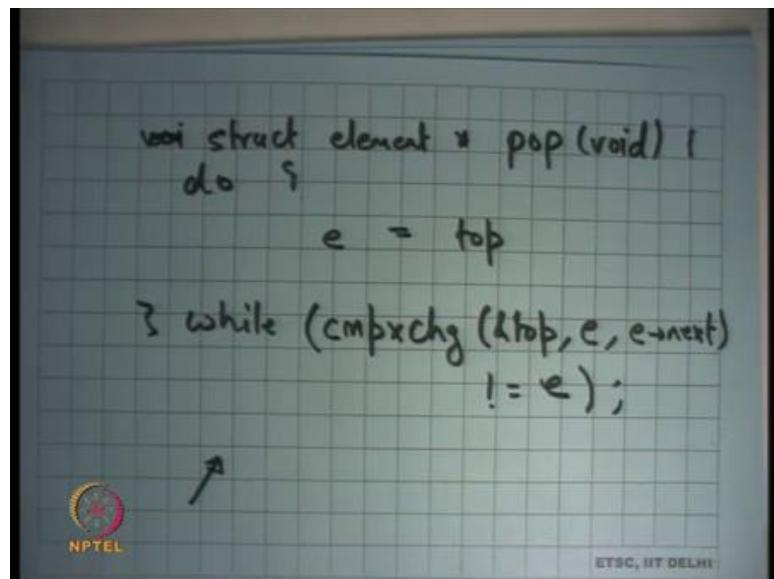
You check atomically if the value of top is still equal to what you read it earlier and if so, you replace it the e right that is what you do right. And, this check with e.next is basically saying whether this was actually successful or not if this was successful then you are done you break out.

If this was not successful which means it was not equal to e.next it was not successful, then you retry this operation. So, let us say this is top I read the old value in e.next this becomes old, then I execute CAS on top and old and the new value that I want is e right.

I want to push the elements I want top to become e so and so I do execute this CAS and top old e right. This will become successful if nobody else has modified top in the middle.

So, you know your top will get replaced with e. If somebody has modified top in the middle then this will become unsuccessful right and the way to check whether it was successful or unsuccessful is to look at the return value that is all and you keep trying until you become successful right. So, this is this how we have to push and similarly you can do pop right. So, I am going to write pop you know you can just this piece of code and pop is very similar. So now, let us say let me just write pop because you know it is going to be important in the discussion that follows.

(Refer Slide Time: 39:17)



So, let us say l struct element star pop I declare a local variable e is equal to top and then I tried to compare an exchange top. The old value that I have read is e and I replay want to replace it with the new value which is e.next right e.next right. That is what I want to do, and I want to do it in an atomic fashion, it is possible that this operation fails how do I know whether this operation has failed. It has failed if the return value is not equal to e right that is it and you put it in a while loop and that is your pop right.

So, that is your push and pop, what are you doing you basically making sure that atomically the list gets updated either with a push or a pop. If I do that what happens to search, do I need to can I say that now search can execute in a completely

unsynchronized way, search does not need any synchronization. Well I mean what are the guarantees that I want to get to the user.

I want to get to the guarantees that the user that my data structure always remains well formed right and my search always sees a consistent data structure, which means if it starts from a top then it sees a location, that now there is the serializability in the access right. So, by serializability me it means there were about ten operations that are given to you, then the final results that I get for all those ten operations look like some serial order like it seems like there is some the operations were done in some serial order.

So, either in a push was done first and then there was a search then was a pop then there was a search and so on or you know something. So, there must be some serial order they should be so that is what the guarantee I want to make. So, the whatever final result I get it should obey some serial order there should be some serial order, that basically will give the give you the same result as you as what you saw.

So, in this case because you are automatically updating the pointer the search if it executed before the compare and exchange will see a list where the push had not been made and if it executed after the compare and exchange it will see a list it such that the push has been made right. So, it serial it is the serial order depends on whether the search executed before or whether the search started you know whether you started with the top pointer before the push or after the push.

So, when you read the top pointer, whether that read of the top pointer was before the push or before the compare and exchange before the CAS or after the CAS right, that is what we are going to make it serial and that is going to define the serial order. So, if it was after the CAS then it is as though the search executed after the push, if it was before the CAS it is as though the search executed before the push ok. But that is not all so but let us say you know; I am holding a pointer to so firstly you know let us see what are some problem with this kind of a compare and swap.

Firstly, why was I able to do this, I was able to do this because my data structure was such that updating it required one pointer swap right. I was pushing involved swapping between e and e.next and popping involved for invading e.next and e you know something of that sort and so I could do this.

But if my update operation required swapping two pointers or three pointers or n pointers then I cannot use CAS right then I use need. So, let us say for example, I want if I wanted to support another operation it said remove an element in the middle of the searchable stack right. So, I give you a pointer inside to an element inside the middle of the stack and I say remove this pointer element.

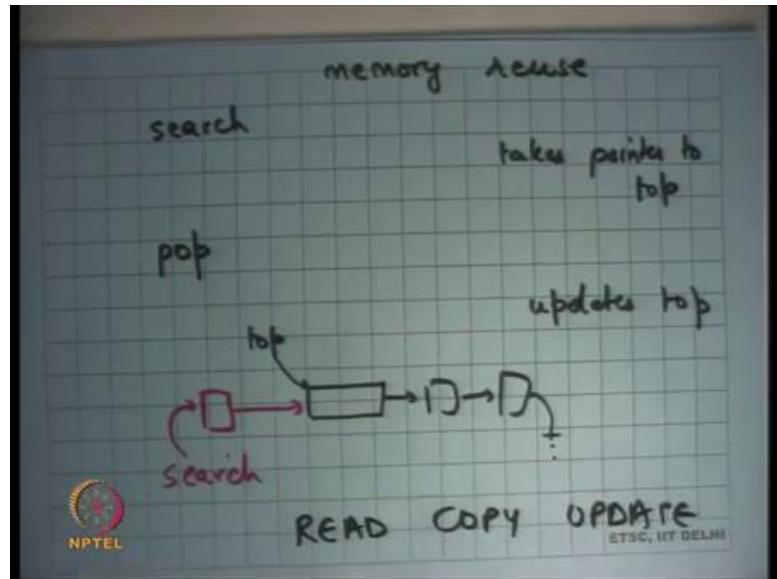
So, removing that element would involve changing the pointer from you know let us say w link list, then I would need to change the pointer from a previous node to the next node and from the node next node to the previous node. So, I need to make two pointer changes right and I cannot do it atomically using the CAS instruction. If my hardware in supported a double CAS instruction which in what they took 5, 6 arguments, then yes or 6 operands.

Then yes. I could have supported something like that right x86 or you know in general architectures do not support more than one memory location for CAS. So firstly, you know CAS can be used for if the operations involve only a single pointer update and one needs to be very carefully reason about it. So, that is one disadvantage the yes.

Student: Sir, in this case say if the pop is not a free the pointer

If the free the pointer right good. So, the second problem is memory reuse alright. So, let us say I popped a pointer, so I popped the locations on the stack right and there was a concurrent search that was running. So, let us say let us draw a timeline.

(Refer Slide Time: 44:41)



So, search gets to execute takes pointer to top right, then pop gets to execute updates top right. So, we said it is because you know it is as though the search executed before the pop and so as long as the top pointers will still exist, and the top is still pointing to the disk.

So, what will happen is at this point you will have a list where the top is pointing like this and but you also have search holding a pointer, search holding a pointer to a location that has pointing to this right. So, that is what that is what will happen. Search is holding a pointer to something, but you know, and you have popped it off, but you are still holding a pointer.

And you say it is because you know this location is still pointing here and you know when the search is going to execute like this it is still going to see a consistent list. So, it is ok. But it is if this location is not freed right if this location was freed by the pop then this pointer can become invalid right. So, memory re-user is a problem right.

So, I am the search is holding a pointer to a location that is no longer a part of the data structure, search operation will remain consistent if that location is not freed right. So, we have a strange situation there we have a pointed to us location that is not really a part of any global data structure. But yet I do not want it to get freed because some local references to it may still be hanging around.

So, the short answer to the question whether I need to do something to search is that well I mean search will work just like that except that you need to be careful about memory ok. So, you cannot just free the location. So, question is when can you free the location, you are executed popped this location is no longer used when can you call free is there a guarantee to when I can call free.

Student: If there is no if there is no pointer which references it.

If there is no other thread that holds a reference to this location how do I know if there is any other thread that is holding a reference to this location.

Student: Sir again this struct element we can keep a counter of how many location.

Wonderful, so here is an here is a suggestion that let us put inside the pointer inside the struct element, let us put a reference count right which says how many people have hold I have holding a reference to it right is that a good solution.

Student: (Refer Time: 47:41) which lock kind of.

Yeah, it will have the same cache line for bouncing problem right. So, each time you search called search you have to increment the reference count and so you have made a write operation out of a read operation. And, so reference count is a possible solution, but it is you know it brings us back to the same issue that this cache line bouncing with the reference count right.

So, the answer is really I cannot I cannot have an you know, if I just want to if I do not want to have read write if I do not want to have a write operation global write operation. Then it is very difficult to know whether there could be any thread holding a reference to a location that that is no longer part of the data structure.

So, I do not have any answer and when I can free it well. I could say something like the following though I could say you know let us wait for one hour and then free the location right. That makes sense because after all if there was a search procedure that of holding a reference to it, it must have you know gone you know we have moved on with it you know how long can it hold the location to the search. But, that is not a good answer right because after all it can hold a reference to this location for 1 hour, you know there is no

guarantee that it will not leave that location after 1 hour and that how did I choose one hour why is 1 hour is a good numbers etcetera ok.

So, we are going to see you know we are going to see a solution to this problem in the context of the Linux kernel, which is called read copy update and we are going to discuss that next time ok.

So, let us stop.

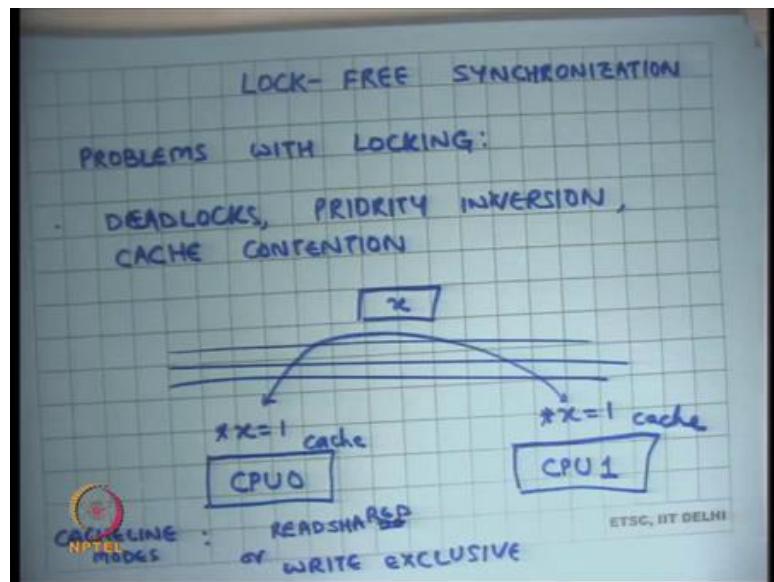
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 39
Microkernel, Exokernel, Multikernel

Let us start. So, a cache line can be held in read shared mode across multiple CPUs or write exclusive mode. So, the idea is if you if the CPUs are only accessing the location in read mode, then it can exist in multiple caches at the same time. If it is being accessed in write mode, then it has to exist in one cache at any time right.

And if some other CPU accesses it while this is cached in some other CPUs cache, then the cache line has to be brought from that CPU to my cache, that CPUs cache to my cache right. So, that is basically write exclusive mode. And so, what happens is if there are two CPUs which are accessing the same location in write mode, then there is a lot of cache line bouncing that is going on.

(Refer Slide Time: 01:16)



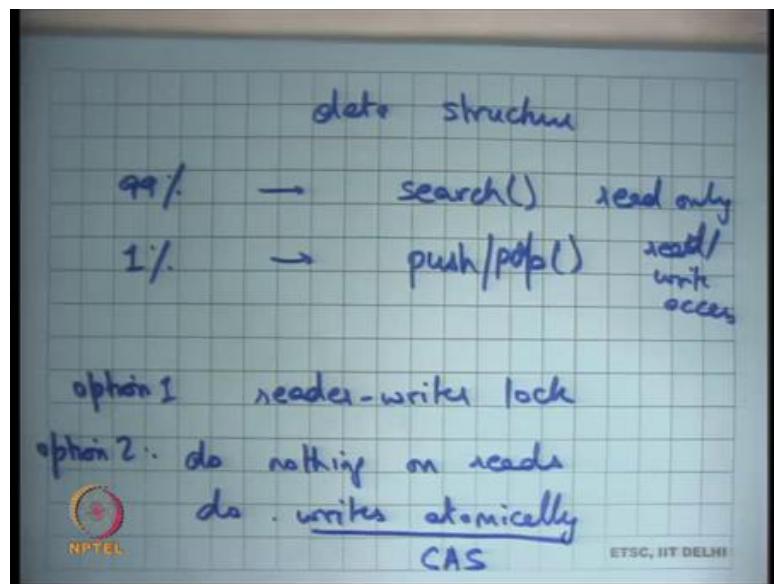
And this cache line bouncing is not limited by the speed of the CPU, it is limited by the speed of the bus right; at what rate can the bus do these transactions of cache line bouncing. And so, what can happen is that most of the time the CPU is just idling waiting for the cache line to come to it.

On the other hand if all these accesses were cache hits, then the CPU could have executed at full frequency and could have been you know up to 10 times faster depending on what the workload is, but you know a cache hit versus the cache miss can have the up to a 10x 10 x penalty ok.

So, what would have been a better thing to do? A better thing to do would have been if I could organize my code such that most of my code is read accesses and so, all the shared variables will get cached in read shared mode in all the caches and all the CPUs will execute at full speed right.

We also said that because of the cache line bouncing problem, it can so happen that your code with two processors or two threads is actually slower than the code that could that you know that was just running with one thread because you know you are not using your caches effectively ok.

(Refer Slide Time: 02:28)



So, we said let us take a data structure and let us say you know 99 percent of times you are accessing you calling search on it which is a read only access and then you know 1 percent, you are calling let us say push and pop right which has read write access.

Ideally you would have wanted that your search operation executes at full speed and by full speed, I basically mean that your data structure should get cached in the local caches of each CPU and read mode; you know that is search is executing most of the time and

so, all these CPUs should execute at full speed. However, you need to make sure that you know search is properly synchronized with write operations like push and pop and so, you know one way to do that is basically have a reader writer lock.

If you have a reader writer lock, then each time you call search you have to set the state of the reader writer lock to locked and that becomes a write operation. And so now, you are not bouncing on the cache lines that hold the data structure, but you would be bouncing the cache lines that hold the lock right. And so, that is not a; that is not you know that is not a particularly good performance and it is possible that the performance actually worse than what it could have been on a single processing.

So, the better thing would have been to you know do nothing on reads. Hey this is option 1 let us say and option 2 is to do nothing on reads and do writes atomically all right. So, what does this mean? Reads do not need to do any synchronization. So, read the code for the search procedure remains the same, you do not use any locks for that. But in your right in your push and pop you will ensure that this update operation happens in one shot atomically right.

So, the read operation or any other write operation cannot see the update operation half done all right. So, what this will ensure is that if there was concurrent reads and writes, then either the read could have started before the update or the write could have started after the update, but the read could not have been I have started in the middle of the update with the update is atomic right.

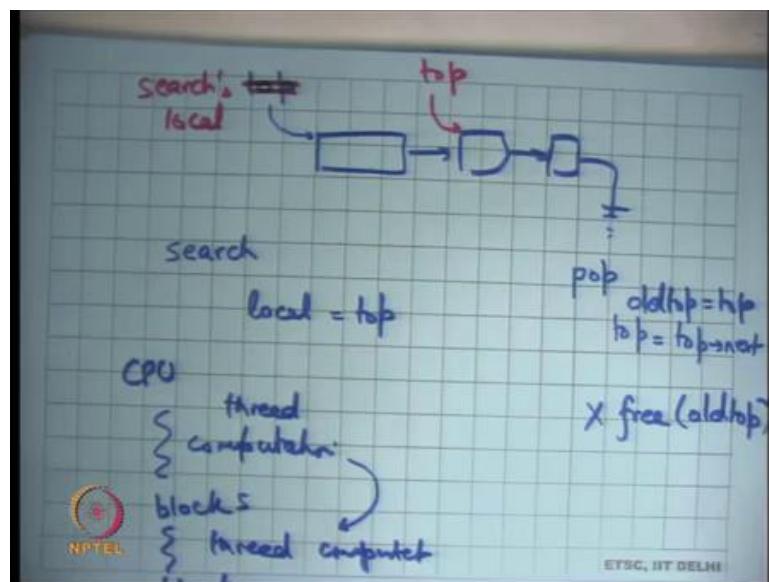
If you could do that, then your reads execute at full speed which is 99 percent of your time and your writes execute you know roughly at the same speed, it does not matter; it is anyways 1 percent of the time ok. So, how did we do this? How do you do writes atomically? Well we use this compare and swap instruction on the hardware that allows you to atomically update 4 bytes right. And we said this is not always possible, it is possible when your update operation involves update of 4 bytes.

So, the way you do it is you first in local memory construct you know you first copy the data structure copy some parts of the data structure locally, compute on the local copy and then you use the compare and swap instruction to update a data structure in one shot right. And so, it is also called read copy update, you read the data structure or you read some part of the data structure let us say you read the top pointer or the head pointer, you

manipulated it you change head to head you knows head to something else if you are doing push or if you are doing pop, you change head to head dot next in whatever case.

And then you use one instruction to atomically swap head and head dot next right and so, that it is read, copy and then you atomically update ok.

(Refer Slide Time: 06:30)



So, we were looking at this last time and we said let us say here is my data structure right and let us say this is my top and a then somebody called search and he took a reference to top. So, he said local is equal to top right. So, local is some local variable that search is holding; it could be a register for example, just allocated in resistor.

And simultaneously, let us say there was a pop operation and so, pop dot top is equal to top dot next. So, what will happen is that top will now point here, but search will hold a reference to the old top ok, but we say it is because even though search executed after pop has happened so, search will still see a well formed list and so, it is as though search happened before the push about the pop. The only issue is that you should not be reusing this memory right.

So, after you have after a thread has popped the first element in the stack, it will probably want to free this memory right. So, after so somewhere here, it will want to say free you know whatever the old top was. So, let us say old top is equal top. So, you free old top. Now there is where the problem is right. You cannot just free the old top anymore

because you do not know whether there could be a concurrent search that is holding a reference to this old top; question is how when can I be sure that I can free it? And the answer is really I can never be sure in general right.

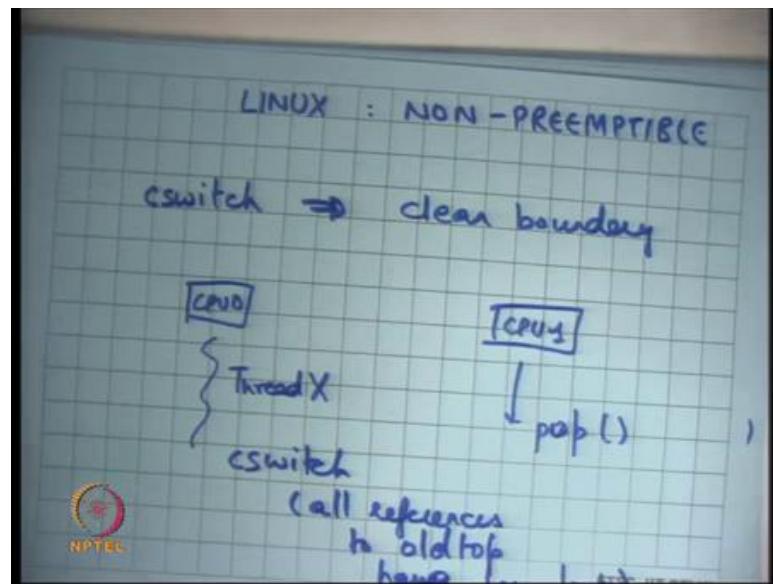
I do not know how long this other thread that called search is going to take to actually start dereferencing it right. Search is going to have a copy of local, but probably you know as soon as it is executes; the next 10 instructions, it will probably lose this reference. But we do not know when it will execute these 10 instructions and we do not have any idea of who all are holding this local this reference to the old top.

So, because this pattern was very common in the kernel where you have a data structure that is being read a lot and updated only very occasionally maybe once in hour or once in days or something, but it is being read a lot; it was very important to optimize something like this ok. And so, the solution that was proposed was to look at the structure of the operating system. Operating system goes through cyclical activity all right.

So, what happens is it does some computation and then blocks. If I look at every CPU it runs on thread, then blocks let the thread computation blocks, thread computation blocks. If I could be sure that before blocking, all local references to any shared data structures have been given up that would be helpful to me right. If I could be sure that across this blocking, I could not be holding a reference a local reference in my stack or register to something that is global, it would have solved the problem for me right.

Because what I would have done is I would have waited for all CPUs to reach this block and once the CPU has reached a block, I can be sure that you know it is not holding a reference to old top right. All future accesses to the stack will start at new top, it is only the old ones that could have old top. But if it has reached the block, then I can be sure that it is not holding a reference to old top and this is true for the Linux kernel.

(Refer Slide Time: 11:12)



So, let us say. So, Linux kernel is non-preemptible which basically means that if a thread is executing within the kernel and there is a timer interrupt or any other interrupt, it does not cause a switch of it does not cause a context switch across threads right, it waits for the thread to you know yield before it actually does a context switch which basically means that if there is a context switch from thread one to thread two implies clean boundary. Which basically means whatever was the operation that the thread had to do inside the kernel that has been completed before the context which has happened, you cannot context switch in the middle of an operation all right.

So, what this also means that if there is a context switch, then I could not be holding a reference a stale reference to a shared data structure in my local variable across a context switch right. In other words, the search procedure could have would have either finished completely before the context switch or would be started after the context switch it cannot happen that I am in the middle of the search procedure and a context which happens ok.

Given this information, I can devise a strategy I can say that if there is a push or a pop, I do this atomic update right I do this atomic update and I wait to free a location till there is a context switch on all the other CPUs.

And there is a context switch on all the other CPUs, I can be sure no other CPU is holding a reference to this location. I can also be sure that any thread that is not running

currently on any CPU, definitely it does not hold a reference to old top because if a thread is actually blocked, then you know it cannot be in the middle of the search it has to be either after the search or before the search.

So, this period where you wait is called the grace period and it depends on the system how you implement the grace period on something like the Linux kernel. You could wait for all the other CPUs to do a context switch before you can be sure that I can free old top after that all right. So, the algorithm is that push or pop or any update are going to atomically update the data structure in a read copy update manner and then I am going to wait for all the other CPUs to call a context switch.

And once they have called the context switch, I can be sure that I can safely free this old top. How do I wait for all the other CPUs? Yes question.

Student: wait till this processed because like one CPU context which is, and the other normal process can also use the same barrier.

Ok.

Student: There is one context switches that all (Refer Time: 14:35) all contexts which use.

Let me just try to understand what you are saying. So, you are saying that let us say there is CPU 0 and there is CPU 1 right. This one says push well let us say this one says pop and he has not freed the location. So, question is when can he free. So, let us say there was some other thread that is running on the CPU; let us say this is Thread X. So, my first assertion is the only other thread that could be holding a reference to old top is Thread X, there is no other thread that could be holding a reference to old top ok.

Because if there is a thread that is swapped out that is not currently running that is definitely not hold that is either you know finish with search and or I it did not call search at all and if it call the search when it comes back again it is going to restart you know starting from the new top. So, the only other thread that could be holding a reference to old top is Thread X.

Now if you can generalize that if the another CPUs and another threads that could be holding a reference of thread top. Question is till when can these threads hold a reference

to old top? So, my suggestion is that because Linux kernel is non preemptable; if there is a context switch on all these other CPUs, then I am sure that after the context switch nobody can be holding a reference to old top, make sense right. So, if there is a context switch here, then I am sure at this point all references to old top have been lost.

Student: Sir.

Yes.

Student: Is that one of the CPU take very long time to context switch and then it might slow down the rest of the thing?

So here is an interesting thing, it is possible that one of the CPUs takes a very long time to context switch and so, it slows down everything else does it really slow down everything else? What am I waiting for context switch what am I waiting to do before all the other contexts which is happen?

Student: Free.

It is just a free. So, if it takes a long time for all the other CPUs to context switch then it is a just means that that location lingers are on for a long relatively long time, but that is ok. Assuming there is a lot of space there is a lot of memory, you are just holding on to a location for a longer than it was needed. Assuming memory is plentiful that is an extremely useful trade off to have.

Student: Which let us say the change that we made is going to be used for CPU one has call pop when normally change that is been made will be used by the later process and till the free command is called probably you cannot start.

So here is the question if one thread is called pop, he has made some change so, can the other threads use that change before the free has been called?

Student: No, it is not.

Why not?

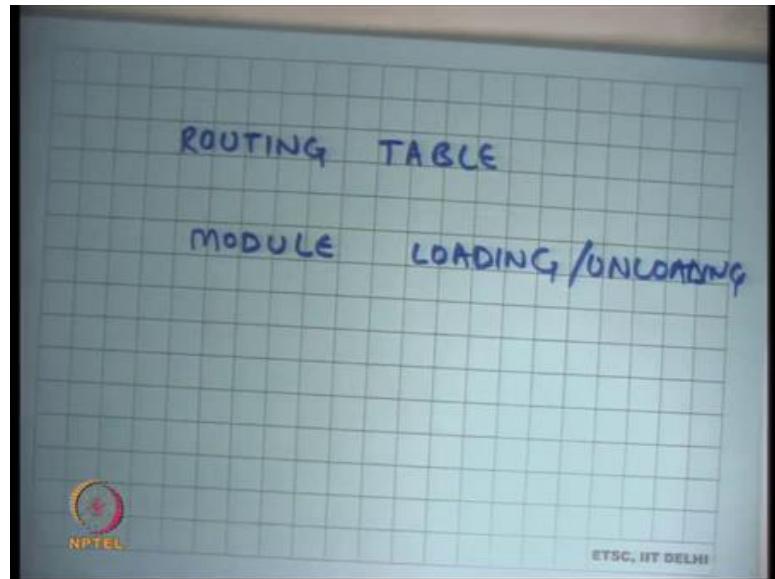
Student: Since because they have started before the change has been made and so, it may cause look at the ... if the ... benefit (Refer Time: 17:59) maybe loss.

I do not see why. So, my suggestion is that if see as far as the pop is concerned it has atomically up made the update, it is just a matter of freeing the location and freeing can be delayed arbitrarily. Assuming I had infinite memory, I could I would say you know I do not even need to care about all this. Assuming I had infinite memory, I would just, you know just execute cast to implement pop and I would not even worry about freeing and I could be sure that so on.

So, it is in a way RCU is a way of trading space for time, you know you are using extra space for longer, but that way you basically make sure that your reads are very fast you do not need any synchronization on the read path all right. So, it is a very interesting idea and it is very practical and useful and let me tell you a few examples there is actually very useful in the Linux kernel. So, firstly, notice that in this RCU approach I use some property of the software in this case I use the property of the Linux kernel that it is non preemptable.

So, I need to know this kind of information to be able to and you know implement this kind of thing. If I could not make any suggestions about the software in which this RCU was implemented, it has it will been not possible to implement the scheme.

(Refer Slide Time: 19:24)



So, what is the property that I am looking for? I am looking for some cyclical activity and I need some point in that cyclical activity where can be sure that is a point where you know all old references have been lost local references will be lost right. So, that is good

enough for me. So, typically you will implement things like RCU in the kernel where you can make these assertions.

RCU at the user level is much you know much more complicated I do not know I cannot just implement RCU as a library, like I can implement pthread locks and other things right. Pthread locks make no assumptions about the application behavior RCU makes assumptions about the application behavior. So, you know there are user level libraries that implement RCU, but you would tell them what is what the psychical activity is and when is it safe to actually you know release references or free things in a safe manner ok.

But let us look at the Linux kernel where RCU is used a lot and where is it used? It is used for example, in the routing table. So, routing table is an example of data structure that has a very high rate of lookups and each time a packet comes you look up, the routing table to figure out where it should be sent and assuming you have multiple network interfaces to this machine and it is acting as a router.

So, there is a routing table very occasionally do you update the routing table, but you need to make sure that you know these updates to the routing table and the lookups to the routing table need to be synchronized with each other. If I had used read write logs to synchronize this, then I would have severely penalized my look up path which needs to be very fast because of a cache line bouncing problem.

On the other hand if I use RCU you know that makes my lookups full speed and my updates are relatively slow in the sense that I cannot reclaim memory immediately, otherwise updates are as fast as they could be; it is just the free that becomes low ok.

Similarly, Linux has what is called modules right. So, what are modules? Modules are fragments of code and data that can be loaded into the kernel at runtime. So, you can compile something and then you can load it at runtime and there are some interfaces it is fine. So, it can sort of start running in kernels space. So, those are called modules for example, you would run device drivers as modules.

If you have a new device you basically attach it, your kernel does not have device driver for this you, you implement your driver as a module and you load it in the module

will though the difference between a module and a process is that a module executes in privilege mode in kernel space you can do everything that the kernel can do that is all.

So, once again module modules are an example they are you will probably be doing lots of module read accesses where you say you know there is let us say some module table that looks up the whether it has this function or not and you are going to call that function. So, there are lots of read accesses and occasionally you are going to do module load and unload right. So, unload and load are very very occasional operations with respect to module accesses read accesses.

And so, here is another example where you could use RCU to execute your read accesses at full speed and yet synchronize correctly with your load and unload. Let us think about XV6 where does it make sense perhaps to use RCU instead of locks and if I was to use it, then what could I have how would I have implemented it?

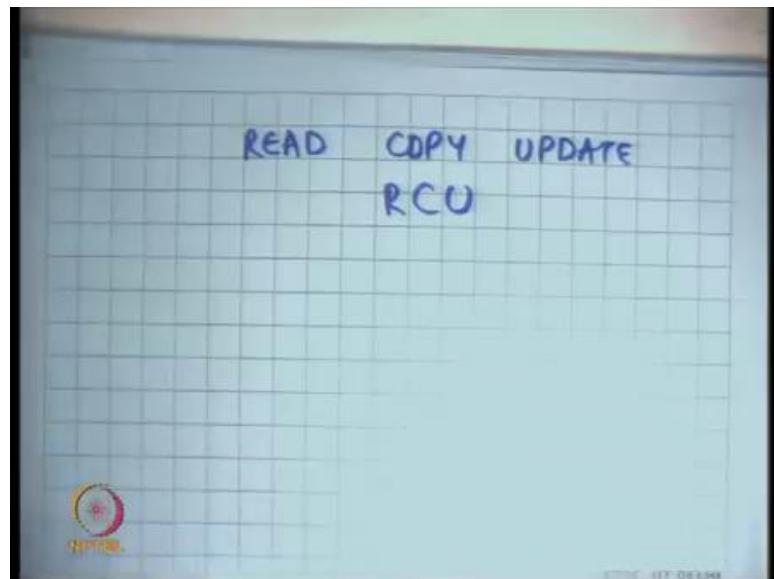
Here is my here is one suggestion. Let us say you know there is some buffer cache entries that are accessed too much in read only mode and not so much in read write mode. So, let us say you know let us leave that question what where it can be used it can be used anywhere where you know there is a lot of read access and I am very few right access; let us talk about how you would implement it ok. So, everything else is I mean it is basically saying you use cache instead of raw logs you do not do anything when you are in read path.

The question is how do you decide what is the cyclical activity all right. So, in case of XV6 if you remember, you can probably say something like this that anytime a thread. So, first again same just like before if a thread is context switched out, you can probably say that it is not holding any references to share data structures because there was context which doubt it has voluntarily called the yield function right either because of the timer interrupt or something else. And you could probably I mean you could organize things in such a way it will never called yield without actually you know coming out of any access is of the shared data.

And then you can say that any time and yield internally would call let us say wait and so, you could you know synchronize that wait boundary. So, you could say a CPU goes through some computation and then wait and then some computation and then wait and so, any time the CPU calls wait you can be sure that is not holding any local references

to shared data. Also, you could say if a thread if the CPU ever goes to user mode at that point, I can be sure that it is not holding any shared references to user data shared data local references of the shared data; good right. So, I have discussed read copy update.

(Refer Slide Time: 25:10)



So, that was the it is commonly called read copy update or RCU for short all right.

(Refer Slide Time: 25:26)



With that I am going to move on to my next topic which is OS organization. So, far we had been looking at the Unix model of an OS and this model also called a monolithic kernel.

Student: Mono.

Why is it called a monolithic kernel? Well it looks something like this, there is let us say kernel space and there is user space and you know their user processes is which have separated at this face, but the kernel is one giant blob single address space and within this single address space, there are all kinds of module like file system, virtual memory, scheduler, drivers and so on ok.

So, this is one big sort of one big blob and that is why it is called monolithic ok. This is this by far one of the most popular models of an operating the organizations of an operating system at least till they stop in server systems and that is the model that you for example, used in modern operating systems like Linux windows BSD etcetera ok.

What are some bad things about this kind of organization? Firstly, because they are all sharing one address space and one protection domain; if there was a bug in a device driver, it can actually bring down all the other things right. If there was a security hole in your driver device driver, he could use it to look at your files he could use it to look at your virtual memory subsystem. So, there is very weak isolation between all these different components and so, it can you know gradually become really large this whole monolithic kernel can become really large especially the device driver because there is so many devices they become you know they become a problem.

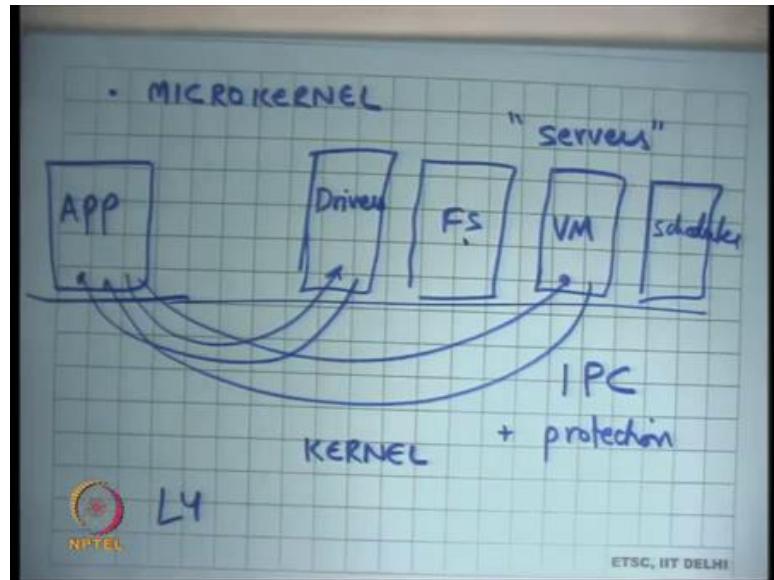
The other issue is that performance has to be tuned you know you have to work for with an assumption that no so, there has to be some file system here right. So, let us say the file system that we discussed is ext 3. So, how do I choose whether ext 3 is my right file system or whether something else is the right file system? Well my choice will depend on what kind of application they are going to run on this file system wait.

So, I have to commit to a type of file system, or I mean of implementation of type system file system very early. I have to guess what is going to be run on this operating system at the time of OS design and that is not a you know it would have been better if a user had complete flexibility to say I am going to run a database application. So, ext3 is not the best file system that to have.

And so, you know here is my file system that you can plug into your operating system. So, you know there would be more sort of customization to applications in terms of

performance and even functionality. If, but that is not possible in this monolithic model the best you can do is support a (Refer Time: 28:50) few different file systems like you know ext3 and tfs or something, but that is not; that is not flexible enough ok.

(Refer Slide Time: 29:05)



So, there is another model called the micro kernel which was made popular in the mid nineties and the idea here is that you would implement all these services file system virtual memory subsystem, scheduler, truck, drivers and anything else that the operating system provides. I will just save separate servers in separate address spaces ok.

And then there will be other applications that is running, and the applications could talk to these modules like this. The interface for talking to these modules or do these servers is what is called inter process communication we know about this IPC Inter Process Communication right and so, what the kernel is kernel becomes really thin all these different servers that were present is part of the kernel in the monolithic model are no longer part of the kernel.

So, kernel becomes really thin. All the kernel needs to do is implement abstractions that allow inter process communication. For example, they can just implement pipes and be done with it ok. And then there are all these different modules that are running all these different servers. So, that is a micro kernel. Of course, you know you could have privilege levels for example, you could say that the file system server can access your

disk while the regular application cannot directly access the disk. The virtual memory server can access the physical memory, but you know other things cannot access.

So, you can implement. So, the kernel is basically IPC plus protection you know who can do what basically access control that is all. So, the nice thing about this is you know it gets rid of both the problem that I mentioned in the monolithic kernel. If there is a bug in your VM in your device driver, the only thing that goes wrong is a device driver; all your other parts of the kernel remain completely isolated because these are completely executing a different address spaces.

More importantly if I want to run a database application, I can you know choose my file system by just supplying code and you know doing some level of authentication to say that this code is trustable in the terms of access, you know letting it access your to my disk.

And so, you know I can use it. It sorts of also you know fits in with the principle of least privilege that we talked about I only need to give privilege to this particular server to be able to access the disk and nothing else. For example, it cannot directly access physical memory for example, ok. So, these are good advantages. What is the problem with this? Why is it not so popular?

Student: Slow.

It is slow right. If you know the application needs to talk to the virtual memory servers system, it needs to go from user to kernel, kernel to user, then user to kernel again and then back right. So, they are least poor user kernel crossings that you going to make as opposed two in the monolithic kernel case right. So, it is slower and there were lots of techniques that were proposed to make to implement fast IPC.

So, special interfaces were developed and the kernel and there are you know they micro kernels like L4 that implement fast IPC, but it is not; the idea has not caught on as much. It is caught on in specialized domains like embedded systems where protection is very important where reliability is very important etcetera where you could, where you could do this.

Student: Sir, if the virtual or the VM system was basically trying to provide an address spaces to other processes, but now it is itself a process.

Ok.

Student: So, as a how relate as an relate, we need to do bootstrapping.

So, the VM process.

Student: (Refer Time: 33:20).

Or the VM server was supposed to provide address spaces to other processes, now it itself is a process. So, it does not need to be some bootstrapping? Yes, I mean you know you could imagine that this some amount of bootstrapping that is going on and this VM process has a special address space you know which is equal to the size of the to the you know what maybe it is equal to the identity mapping between virtual address space and the physical address space. And then it has special privileges where it can actually make mappings and change the page tables of all the other processes right.

So, that is just protection and access control and you know this process is allowed to change the page table of that process. So, that is basically you know, but you know the nice thing is that different servers. So, I can come to my the operating system and say that has my application and I know that the best possible the best cache replacement algorithm for this application is not allow LRU, it is let us say MRU right.

So, I can just change my VM subsystem you know I can just plug in play and use MRU instead of LRU that is it right similarly I can use instead of logging I can use ordering or whatever you know. So, this plenty of choices what my cache replacement policy, what is my buffer cache replacement policy etcetera, what is my prefetching degree etcetera.

So, it can all tune it and completely configurable. On the other hand, if you use something like a monolithic kernel like Linux, the developer of Linux has pre committed you to a certain algorithms which is which may not be the right thing for you.

Student: Monolithic kernels used to have the (Refer Time: 34:54) all new drivers and so, can be the we are going to can be exchange into.

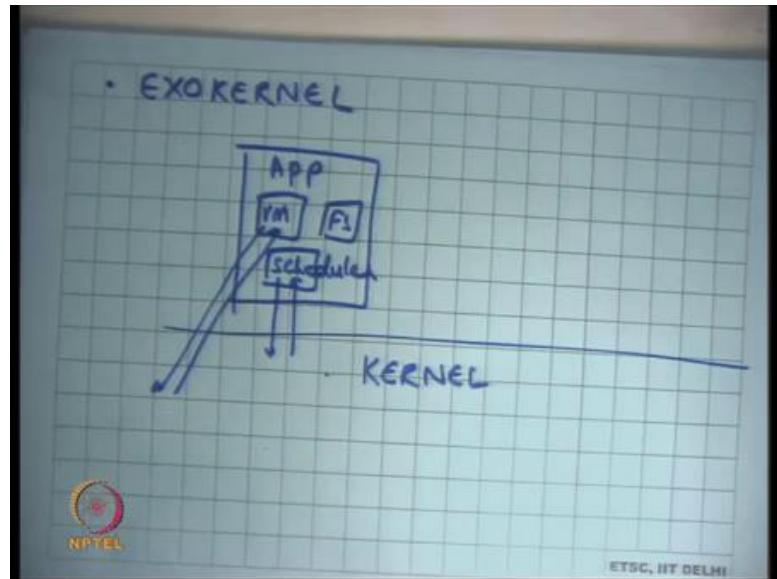
That is a good question. So, in monolithic kernels, there is an ability to install new drivers; for example, and the this ability to install new drivers is what is called through loadable modules. So, in modern kernels you have the notion of loadable modules that is what I was talking about. Loadable modules are nothing, but blobs of code and data that can be loaded in kernel space and they execute with the privileges of the kernel all right and cannot you just use the loadable module idea to plug and play other parts of the kernel like file system virtual memory subsystem and so on.

Firstly, it does not take care of the problem of protection yet right, even the loadable module has identical privileges to everything else. So, there is a bug in your loadable module, it collapse entire system and if you allow arbitrary things to be loaded loadable then you know you that that is the you know you are increasing your surface area of attack so, protection is not handled. And the other thing is that the loadable module interface needs to be very carefully designed. So, that it is rich enough to support all these different subsystems right.

As for example, you know at one extreme that is loadable module could just be behaving like the server in your microkernel where it is just you know the only way to talk to it is IPC, at another extreme you know the interface is so rich that you can actually you know directly you will make function calls to it etcetera instead of doing IPC.

And so, you know choosing that; so, in general while loadable modules have been used for drivers etcetera, they actually also used for file systems really. So, in some sense you know loadable module is somehow somewhere halfway between microkernel and a full monolithic kernel. It is really is giving you some advantages of micro kernels in some sense, but you know there is a protection problem that still sort of does not get solved ok; that is microkernel.

(Refer Slide Time: 37:15)



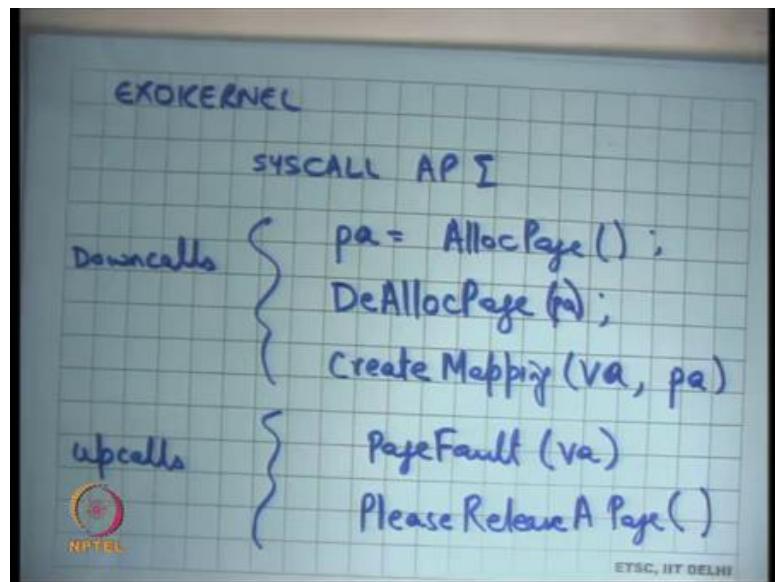
There was all two alternate organization called an EXOKERNEL that was proposed somewhere in late nineties. An Exokernel was also a microkernel except that it tried to move much most of the functionality of the operating system at the application level. So, here the idea is now let us say this is my kernel and this is my application earlier my VM and file system were living inside my kernel instead of that let all these subsystems live inside the application scheduler right ok.

But you know how can they; how can they do that? You provide interfaces such that these different modules can actually function right. So, you basically moving the layer of abstraction one level below ok. So, instead of exposing a file system to the application, you will expose a raw disk to the application and let the application implement its own file system on top of it or instead of exposing an address space to the application, you expose physical memory and page table to the application.

And let the application choose what mappings it wants and what cache will get them it wants. Instead of saying that I am going to give you a scheduler, you have a virtual CPU that is uninterruptible. You export the CPU as it is and you tell the application, there is a timer interrupt you need to choose what you want to do you know you need to relinquish CPU.

And let the application decide what it wants to schedule and what when it wants to relinquish CPU right. So, let me use a concrete examples to show this.

(Refer Slide Time: 39:25)



So, let us say I am using Exokernel example. My system call interface SYSCALL API looks something like this: physical at page is equal to AllocPage then there is DeAllocPage and then there is CreateMapping from virtual address to physical address DeAllocPages you know it takes an argument physical address. These are the down calls I am going to explain what down calls mean and then there are some up calls; up calls are PageFault on virtual address and PleaseReleaseAPage.

So, what I am doing is I am exposing API is to the application to allocate a physical page. So, the application is full control on physical memory or not full, but at least you know better control over physical memory you can say, I need one more physical page. So, it knows exactly what sets current physical memory footprint as opposed to your monolithic kernel where the application was completely oblivious of you know what physical memory footprint I actually have.

All the monolithic kernel application sees an address space. It is the operating system that is playing tricks under the carpet in implementing the data space sometimes it is mapping a virtual page to a physical page, sometimes it is mapping it to a memory mapped file and other times it is mapping it to a swap space right.

So, instead of that give full visibility to the application and let it say I want to allocate a physical page, you know do not play of tricks under the carpet at all; give it the control. And so, it can also say I want to DeAllocate this physical page and you could control

completely tightly how much physical memory footprint it has and then it can create a mapping between its virtual address to its physical address. So, example if it wants to create multiple mappings to the same physical address aliasing, it is free to do that.

So, these are the down calls. By down calls I mean these are system calls that the application can request from the operating system and then there are some up calls; up calls are something like signals in Unix with the operating system can tell the application or ask the application to do something.

So, one other application, up calls is a page fault all right. So, a PageFault up call basically means that the application tried to access a virtual address that it is not currently mapped. So, it has not created a mapping yet for this particular virtual address. So, instead of killing the process or instead of doing things under the carpet itself, it just tells the application that look there is a page for that is occurred.

So, it converts the hardware exception into an up-call signal to the application. The application sees there is a; there is a; there is a page fault on this particular virtual address and that is because I have not created a mapping for it and because that is because I am implementing my own virtual memory subsystem.

So, what it may want to do is allocate a page, create a mapping and then re execute that instruction return from that signal right. So, it is giving full control to the application it can choose which page it wants to replace right as opposed to the operating system choosing on its behalf, yes question.

Student: If it does not call lot of (Refer Time: 43:20) like preference since this since my as is call India is now will be (Refer Time: 43:26) yeah beginning time if there will be (Refer Time: 43:28).

So, what about protection? Good question, what is what about protection? What are some bad things that can happen? Well what about how do I prevent an application from just doing alloc page; alloc page; alloc page till it completely exhaust physical memory? Well I can implement protection at that level I can say that you know this application if the I can implement quotas for every application and I can say that I would not give more than these many physical pages to this particular application.

So, it is you know it is perfectly fine AllocPage can you turn minus 1 to say that you know I did not I did not allocate a page for you and you deal with what with it yourself. So, that is your problem right. So, you can implement protection even at this level now right.

Student: Sir.

What about, question.

Student: Sir is actually to go (Refer Time: 44:14) each of them is dividing a different algorithm for VM. Now there is going to be a problem as in suppose one of them like if there is no consistency in the allocation (Refer Time: 44:29).

So, question is if each process implements its own cache replacement algorithm, would not there be a problem? I do not see a problem right. Why is there a problem? You are just saying that what I have done is I have basically decided I will give this pool of physical pages to you that pool of physical pages to you, you can choose how you want to use this physical pool of pages. It is your decision; you can optimize based on your application which pages to bring in physical space and which pages to keep in swap space that is your decision all right.

So, multiple algorithms cache replacement algorithms can work simultaneously tune to your respective application. Let me give you a concrete example let us say I am a database and I am running the database as a user level application. A database will implement some kind of a cache for all the disk blocks right and so, this if this disk block cache is implemented in the virtual address space which it will be then you know the operating system has no visibility that this is actually cache pages.

And so, it will treat these pages as regular pages and it will keep swapping them in and out of the swap space, but that is a really poor thing to do because the database was using them those memory virtual address space pages as cache slots. And now the operating system ends moving cache slots and you know doing really hard work to maintain consistency of those cache or correct behavior of the correct contains of those cache blocks and moving them in and out of swap memory.

Instead if I had told the database server that I need some memory back from you the database would have known that these are my cache blocks and I can throw them at any time is. They just cache copies, I do not need to preserve their contents because the contains are already present in my disk blocks right. So, that way you know there is a you can save lots of extra copying between your database disk and your swap disk ok.

Now let us talk about the protection bit a little more. So, what is the other thing that can happen. So, one thing is you know an application just calling allocate page allocate page etcetera. Well it is an application is running happily it has lots of pages and then there are lots of other application that start simultaneously, and they also need pages.

So, now, at this point the operating system would want to take pages away from this already running application. In the monolithic kernel it was very uncivilized because I would just you know take pages from you and you would not even know and we said that is not always a very good thing right because the application you could you know application knows which pages to give. In this case, you will make an up call saying please lease a page. So, here it is a different thing you know being more civil you are saying please a page right.

Now, the application is going to decide which page to release. Application knows exactly how many pages it has and etcetera and it is running it some algorithm to say it is going to. But what if the application does not honor your please right you just say is you know the he takes you for a ride he takes the kernel for a ride. So, that is a protection problem I want a page for other applications. I am requesting this guy for a page, but he is not giving it to me. So, the solution to that is you know.

Student: Kill.

Kill or you could say you first you know you add one more up call here ForceRelease; ForceRelease the page right or you know just do whatever you were doing in you know monolithic kernels. You first ask the application please; you know wait for some time and also maintain some history on it is reputation or whatever.

And then you can if it is not doing what it, but you wanted it to do you could probably just take pages away from it. So, you know you could implement protection orthogonally

to your common interfaces and so, I get both protection and performance at the same time. I will continue this discussion next time.

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

Lecture – 40
Virtualization, Cloud Computing, Technology Trends

Welcome to Operating Systems lecture 40 right. So, we were discussing operating system organizations and let us review it.

(Refer Slide Time: 00:33)



So, they are you know there is a monolithic OS organization that we have been used we are used to which is which has you know let us look at the Unix abstractions, the file system, virtual memory, scheduler drivers.

All live in one common address space that is called the kernel and then these different modules exporters system call APIs like read write open close M map, page fault etcetera right. So, these are all sort of attractions that these things provide. Then there was an alternate organization which is called the microkernel, where these different components are moved in separate production domains.

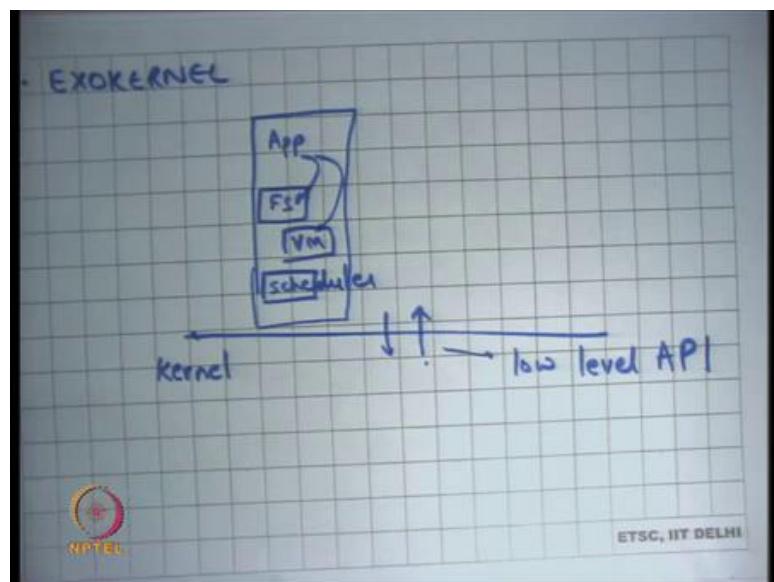
For example, they are run as separate servers; the servers could be running as separate processes in unprivileged mode or they could be running as separate processes in

preparation mode, in either case they are isolated from each other. And, but the kernel is basically it is just an inter process communication IPC and protections.

So, nobody can run away with the resources and it the job of the kernel is really to provide fast inter process communication right. So, this are the advantage that it is very plug and payable pluggable and playable, you can choose your file system, you can choose your virtual memory, subsystem depending on what you want to run, you can actually say here is my virtual memory subsystem why do not you run this one, instead of that one.

Of course, there are some issues there that you know that virtual memory subsystem should be trusted maybe signed by some certificate authority etcetera. But but it is possible to do this right. The other thing is of course, if there is a bug or there is a security flaw in one of these things, let us say there is a security flaw in the driver, it does not affect the security of the full system, it just affects one container and so, faults are not propagated ok.

(Refer Slide Time: 02:25)



And, then we were looking at another type of micro kernel that is called the exokernel. Here the idea is that you expose as much information low level information as possible from the kernel to the application. And, all these different subsystems like the file system virtual memory scheduler, even the drivers are part of the application logic ok.

So, for example, the application can decide what the application knows, that there is a physical address space and there is a virtual address space, and it can decide what should be the footprint of my application or my code on the physical address space, and what should be the footprint in the virtual address space, and what should the mapping be? In other words, it can decide what the replacement algorithm should be when should I take a page fault etcetera and so on right.

Similarly, in the file system case it can choose what file system it likes. So, instead of exposing open rewrite close calls to the application, how about exposing the device interface sort of like the disk interface to the application right. So, tell the application that look you have access to a disk a raw disk and so you can write to sector number 10 and so on.

And, you choose you know what layout you want on this part of the disk. Of course, that has problems you know what about sharing etcetera, but those can be you know built on top of that. So, how do different processes share files etcetera, that can be built on top of that, but at the end of the day the kernel is not involved in implementing the file system. The kernel is only involved in providing an abstraction, that allows the application to build a file system on top of it right.

And, so that makes things very configurable and application can choose what file system it wants, and I was to use the example of a database, you know a database does not like. So, for example, a database may one strong system guarantees. So, if it is written something to a disk block, it needs to be sure that the disk block is actually present on the magnetic platter before it actually you know prints some message or releases the money or something like that right.

So, these kinds of guarantees if you provide on top of a regular operating system like Unix, you will need to do system calls like `epsink`. And, those because there is multiple layers, then all those layers do not interact very well with each other that overall performance of the system is not optimal, but if you have this kind of a system, then an application can actually tune you know there is no you have gotten rid of the layers, the multiple layers you were saying that the application can choose.

Student: (Refer Time: 05:04) chalk (Refer Time: 05:05).

Chalk.

Student: Yeah.

No, I do not have it I do not think so.

Student: Here is. Here is alright sorry. So, the application can choose what file system abstraction it wants. So, there is only one file system abstraction ok. So, let us look at. So, you may say, but you know that is too much headache for the application developer. Does that mean if I want to implement a small application, I need to implement a full file system and a virtual memory subsystem, before I can get hello world to run, well the answer to that is very simple you just use libraries right.

So, you have standard libraries for example, you have you use you know you have a library that implements the ext3 file system it is a user level library right. That implements the ext3 file system using the obstructions provide by the kernel. Most applications we want to just use the ex 3 user level library, but let us say your application is a database application, it says I do not want to use the ex 3 library, I want to use my own sort of a library.

So, this flexibility is possible, because you are giving lower level interfaces to the application as opposed to higher level interfaces for application, yes.

Student: (Refer Time: 06:23) too much over head like (Refer Time: 06:24) will be having some of the information about.

Would not there will be too much overhead interesting, would not there will be too much overhead why do you think there will be too much overhead, you say why well every App will have it is private copy of the file system code. Every process will have a private copy of the file system data or similarly VM data, driver data.

So, is not that too much overhead. Well you know you could argue there is some overhead yes because, but at the same time you could share these things across process is right. So, processes can actually share address spaces. So, libraries that are common to multiple processes, can actually live in shared space, can actually have one physical address copy. And, multiple virtual addresses spaces can point to the same library copy.

And, these kinds of optimizations are well known, and you already know about them right. So, multiple processes sharing libraries, sharing data are actually pointing to the same physical memory area and so, this space overhead is not much very negligible in fact.

Student: Sir, one more kernel has to keep track of like you said last time for every Apps for bad (Refer Time: 07:34) going running away with the.

Right this is what the kernel has to do is firstly, it has to design a low level API, that allows all these applications to actually control these low level interfaces, like low level devices for example, or low level disk or network or whatever or CPU or memory.

But the other thing it needs to do is basically implement protection. These applications are not trusted, they are not trustworthy and so an application should not be able to for example, take to run away with the resources, or it should not be able to create mappings that are it is not allowed to create, or should not be able to write 2 blocks disk blocks that it is not supposed to write to.

Right. So, those are small things that the kernel will ensure.

So, kernel job is protection and that is poor protection and low-level implementation of low-level API that is all. So, that way the kernel becomes much thinner, than the original thing any way protection was there right. If, you whether you put it a higher level or you put it a lower layer, that will remain in protection or you implement protection at a lower layer.

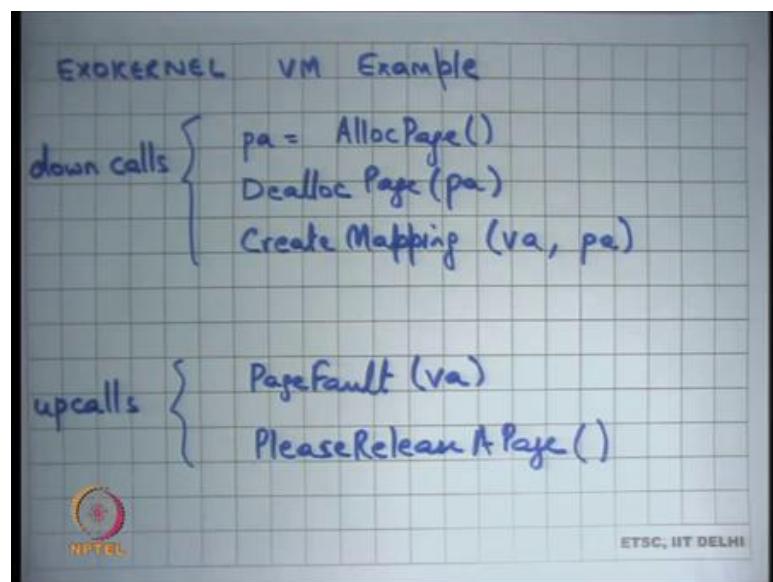
Student: Sir, but in previous case we had like just simpler abstraction that kind of protection for which was same for every App right. In this case we have keep records of which have are (Refer Time: 08:51)

No so, question is in the previous case in the monolithic kernel, the abstractions were identical for different processes. Here the abstractions are not identical for different processes that is not true; the abstractions are still identical for different processes; the abstractions are these low-level APIs. What is happening between the App and the these layers kernel does not care?

Student: So, sir I am saying like kernel has to keep track of whether the app is running for too long (Refer Time: 09:18) did it yield when I requested it to (Refer Time: 09:21)

So, you know kernel has to keep track of how long the App has been running, but that is true even for monolithic right. That is just slightly more complex we are going to discuss it with more examples let us look at this.

(Refer Slide Time: 09:33)



So, we were looking at the exokernels VM example and see said ok. So, because unlike Unix, where the application has no idea that there is something called a physical address space. The application only knows about a virtual address space right. And, it is the operating system that is basically create managing this mapping between the virtual address space and the physical address space. In the exokernel you actually expose this entire mapping to the application, and you can and their interface that you gave is.

So, the that down calls which are you know system calls basically that application can make. So, you can allocate a physical address page and you can deallocate a physical page. So, notice that I am saying pa which basically means physical address. So, the application knows that it is a physical address space. Application has full knowledge about what is the footprint in the physical memory.

What is it is footprint in the physical memory in Unix an application has no idea right, it is OS who is managing how much physical footprint you have and then there is this

down called create mapping run virtual address physical address. Of course, you know there are some production mechanisms and the kernel must ensure, to make sure that it can only create mappings to physical addresses that it owns right, or it has allot right.

So, it is also that there is no security problem of that is all. And, then there are up calls; up calls are similar to signals in Unix, where the kernel can actually ask the application to execute some code, and there is some entry points at the kernel the application must have preregistered and so some up calls are page fault. So, the kernel tells the application that is a page fault on this virtual address and the application can decide that it wants to allot one more physical address physical page and then create this mapping and then resume the execution.

In doing so it may want to first de alloc some other physical address space, because you know the application the kernel is not allowing you to alloc more than certain number of physical address space etcetera. And, then so that is page fault, and the other thing that the kernel will want to make sure ensure is that the application cannot just run away with this memory.

And, so in this case let us say there are other applications that are running, and you want to do some kind of fair allocation of a memory, then the kernel can make an up calls please leave the page. So, notice that this is unlike Unix, where the kernel would just take away the page, it will run some algorithm internally to figure out which page to take let us just take it away right.

Without that without consulting the application at all and that was a problem right. The problem was that I was not consulting the application; the application was perhaps the best judge of which page to actually throw away right and, whether it needs to be written to the swap device or not.

In this case you just tell the application I want you to release the page. And, so, the application then you know chooses which page to release and then calls Dealloc Page on that particular page. And, then you can you know to make sure that this is safe, you can have another Cisco up call which says, you know first release the page and you give a particular physical address that I am releasing this. And, if he should he better honor that, if he does not honoring that, he will probably get a seg fault very soon right yes.

Student: But there might be some algorithm, which are better than at global level. For example, disk scheduler. Now, for example, you have said that in elevator kind of thing in the elevator algorithm, if you have free knowledge of the sector which we need to leave we can persuade a more efficient way.

Ok.

Student: Now in this case every, now in this case every application will be running it is own disk scheduling algorithm.

Ok.

Student: So, it would just be a problem.

So, fair point question is the some algorithm or some policies that are best implemented at global level and not at a local level right. For example, may I do not want to run my page placement algorithm at a global level right, that may be the most efficient thing to do. As opposed to say that, I want to give a take a page from you and you decide which page to give right.

I may want to consider all the pages across all the processes at the global pool and then choose the one that is least recently used let us say. In this case I am you know I am basically pre committing to a process, that this is the process I want to take a page from and then I make an up call saying please release the page.

So, is not there, is not there an inefficiency there well yes, I mean. So, there is a tradeoff. Firstly, notice that there is there are two kinds of policies here; one is the policy that the application will implement internally, and then there is another policy at the OS level. Anyways, which is choosing, which process to ask right.

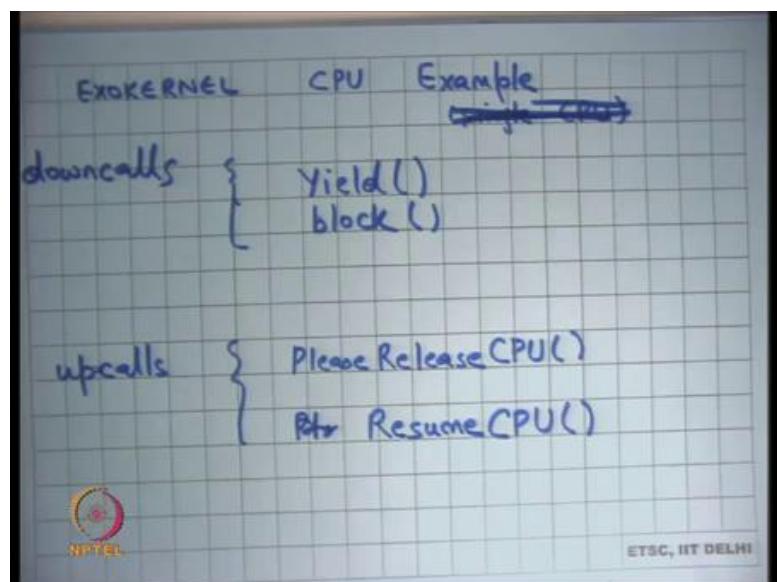
So, it already the kernel knows that each process has these many pages. So, there is there are two levels of policies. And, so there is still a policy inside the kernel that cannot be overridden, which basically saying, which is a global policy right.

So, what the exokernel is doing is separating the global policy from the local policy. Is saying let us let the application have the flexibility to at least decide the local policy,

global policy is still in the hands of the kernel. For example, it will choose which process to give this up call to make, so, that is the global policy.

Similarly, you know any other shared device like the disks you took the example of disk scheduling same thing right. So, there is a global policy engine running in the kernel level, but you are giving the application the flexibility to at least decide the local policy, you are right. The application can still have no control over the global policy perfectly fine, perfectly valid point.

(Refer Slide Time: 15:05)



Let us take another example; exokernel CPU. So, you know you have down calls like yield and block; these are similar to a Unix they basically say I want to yield this CPU. By yield you mean I want to stop running on the CPU somebody else can run, at the same time I am ready to run. So, if there is nobody else to run I can you know I can be rescheduled back and block basically means that I am blocked so, you know you till I am woken up by somebody I am not available to run.

These are similar, but the interesting thing is up calls unlike Unix where, where the kernel would just take away the CPU from the application and we saw some problems with that right. So, the kernel can just say, I need a CPU back just take it away right.

So, irrespective of where the application was in its execution, you could just take away the CPU from the application. And, there was a problem with that let us say the

application was actually holding a spinlock right. And, now within the critical section the CPU was taken away from the application, then all the other processes which were known depending on the spin lock will just waste their time spinning on this spin lock for the time quantum right.

So, that would up that was we used that example at the schedule when we were discussing scheduling right. And, that problem was basically because the application had no control over when there was a context switch. Here you are going to make an up call instead asking the application to please release the CPU.

So, the nice thing about this is when you ask the application to say please release the CPU, the application knows that I am in the middle of a critical section, it can actually execute the minimum number of instructions required to come out of the critical section and then call yield right. That is a much better scheduling paradigm than what we saw in Unix right.

So, it does not have this problem of getting interrupted in the middle of a critical section and so everybody else just keeps wasting the CPU, because it is just spins on the lock. Also, you can you know the application knows exactly which registers to save. So, let us say you know the application is just using 3 registers, so, the when it actually is called yield it just needs to save those 3 registers, before it actually says, I want to yield the CPU.

Similarly, there is an up call called resume CPU. So, unlike Unix where when you resume just start from where you preempted the process last time right. In case of in case in the case of exokernel, you will instead make an up call saying resume CPU. And, so up call is going to go to a certain point in the application, and the resume CPU up call will know exactly which registers to reload.

Depending on which registers were saved in my previous yield call right. So, you have saved on the amount of data that you need to save and restore also.

Student: But the data is being stored by the user program itself (Refer Time: 17:50).

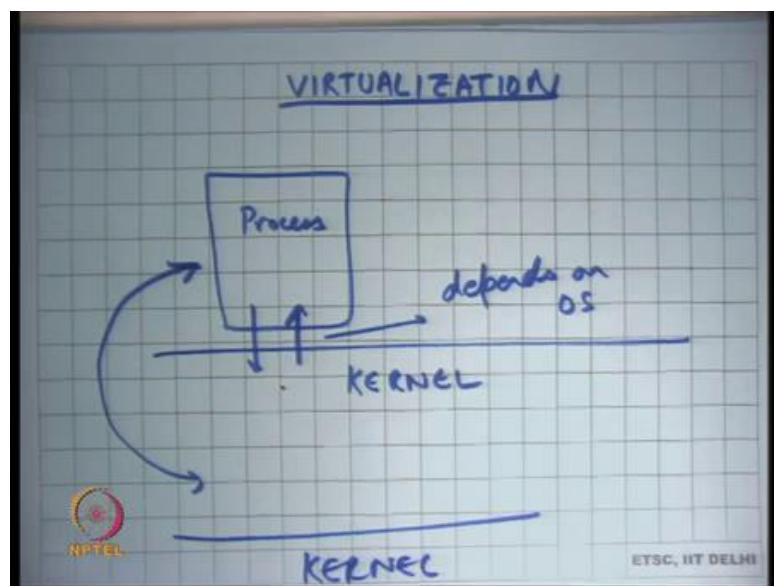
That data is being stored by the user program itself yes. So, the user it is a user programs responsibility to save and restore it is own data right. As opposed to and it has this flex,

you know you can give it that flexibility because you were making, you know you are requesting and resuming through you are giving him control over it. As opposed to Unix where you would just take it away and give it back, in which case it becomes a kernel's responsibility to save everything.

Student: Could it is that much of an advantages, because anyways only a small overhead.

Is that much of an advantage well you know registers are very small it is not a big deal really, but no in certain cases it may be a significant advantage. But, I think the main the most important thing is the scheduling a point that I discussed which is basically, that you do not have problems like the convoy effect where basically you if one process gets interrupted in the middle of a critical section, then there is a very bad scheduling behavior that you get good all right. So, with that I am going to move to my next topic, which is virtualization ok.

(Refer Slide Time: 19:07)



This a somewhat relatively modern topic. Are there practical examples of exokernels? Well so, exokernels were first proposed in late 90s as a research paper. And, you know many ideas from the exokernel paper have been used in modern kernels, especially in the modern kernels where we are basically looking at kernels that scale across lot lots of CPUs, like 60 to 80 CPUs you know these kinds of ideas are actually proving very useful.

And, so many ideas from the exokernel paper are being used in modern kernels. Especially research kernels today which are targeting a large multi core CPU machines all right. So, let us talk about virtualization. So far, we saw that there was a kernel and then there was a process right.

And, there was an abstraction between the process and the kernel. And, this abstraction was designed such that the process can do what it wants to do and also it is safe. This abstraction dependent on the kernel, it depends on the OS let us say. For example, windows will have a different abstraction from Linux; Linux version 2.2 will have a different abstraction from Linux version 3.0 and so on right.

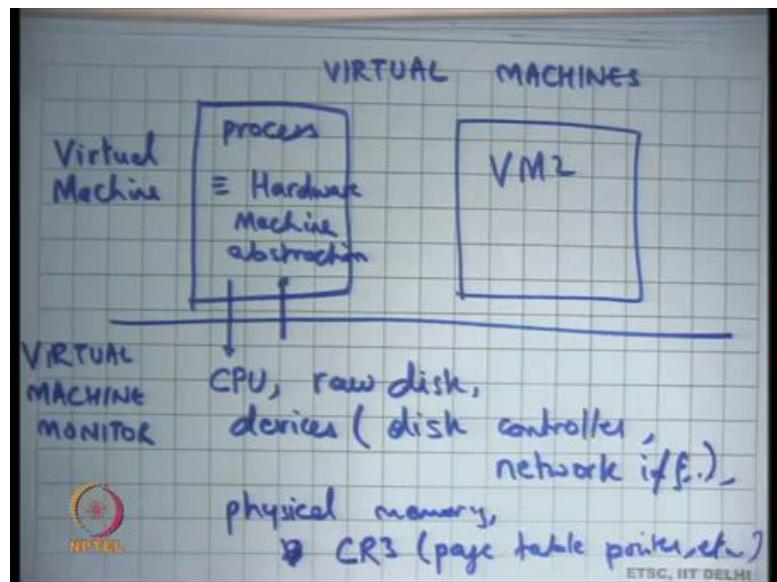
So, these abstractions are relatively very fluid it is very difficult to take a process that would run on a Linux 3.0 and make it run on Linux 2.2. Even more difficult to take a process that runs on windows and make it run on some on Linux let us say or vice versa right. So, all these things are relatively difficult.

Also, it is very difficult to let us say I have 2 machines a process has lots of bindings with the kernel. For example, it has a file descriptor table it has virtual memory etcetera. So, it is very difficult to migrate processes between life between two different machines. So, you have two different machines running, two different kernels; let us say they are even running the same kernel.

You know I cannot just take a process here and just say you know let us run it let us start running it here, you know I cannot take a running process here and then start resume and resuming it on the other side, because a lot of the state of the process is actually inside the kernel. And, there is a lot of state that a process can have.

And, you know as so far, we have seen given that it is a monolithic kernel, a monolithic kernel has a file system, a virtual memory of system and so there are all kinds of data structures that are living inside the kernel. And, so all those data structures need to be migrated as well and that seems like a very hard problem right.

(Refer Slide Time: 22:21)



So virtual so, virtual machines are a concept which says, let this abstraction that I am calling a process be somewhat equal to the hardware abstraction, machine abstraction. If, I could do this efficiently then you know the abstraction that I have with the with whatever is running beneath it let us call it the kernel for now. It is going to be simply that our CPU, a raw disk devices like the disk controller right, network card, network interface and so on.

And, physical memory and virtual memory subsystem, which are whatever the hardware supports. For example, in order all the registers like CR3, which is the which is pointed to the page table right. So, what if the abstraction that the kernel provides to the process is identical to the abstraction now the kernel sees itself on the hardware right.

So, the kernel is returned to assume a certain obstruction which is a hardware abstraction. The hardware abstraction is there is a CPU that runs one instruction after another, it has a certain instruction set jump allows you to change the program counter and so on the memory accesses.

And, then there is a virtual memory subsystem, which can be controlled by going through CR3 and you can configure page tables, there are devices hardware devices that can be accessed using in and out instructions and so on right. There is an interface of the hardware provides and is specified in the hardware manual of the manufacturer.

Can the same interface be the one that the kernel provides to the process? And, if it is able to do this efficiently, then the interface between the process and the application and the kernel becomes much more solid or much more compact, and much more much less fluid.

In the sense that hardware abstractions seldom change or change at a much slower pace than the kernel abstraction change right. You get a kernel version every few months, but the hardware version changes at a much slower rate. Moreover, hardware, preserves, what is called backward compatibility, you know anything that you could have run on a previous generation of a processor, or a previous generation of disk controller will run on the new generation of the disk controller and so on right.

So, if you could do that, then this abstraction becomes relatively solid. And, so it will be possible to take a process and you know run it on a different kernel. And, so you basically know that this state that this process has is very compact, it consists of the physical memory footprint, the page tables, the disk contents and the network interface state and other registers right.

So, these this is basically you can draw a line I mean this is exactly the state of this process and if I can save it and restore it somewhere else, I can migrate this process from one machine to another right. And, I can also migrate it across different versions of the kernel ok.

So, this abstraction is called the virtual machine abstraction ok, because it is just a machine that is implemented in a virtual manner in a process container. And, so you could run multiple virtual machines on one physical machine right, it is just like running multiple processes on one kernel except that the abstraction is has changed ok. The abstraction is not no longer of that of a Unix process, the abstraction is that of a machine of hardware that you are providing it. Does it mean that you are exposing all the hardware to the process directly or not?

So, well I mean; firstly, just like Unix could not trust its processes, we do not want to trust our virtual machines right. So, you cannot just expose the hardware directly. The question is how does one implement these virtual machines efficiently? So, let me just also say that these containers are called virtual machines and the kernel is called the virtual machine monitor.

If, I am doing something like this; firstly, let us understand, what are the advantages of doing something like this? The advantages of doing something like this they are firstly, because the abstraction of the container is very solid, it is very well defined, it is written in the hardware manual. And, it is basically something that has a guarantee of backward compatibility, you can actually take a virtual machine container and move it at will to some other virtual machine monitor and still expect it to run it exactly like it was running in the previous one machine right.

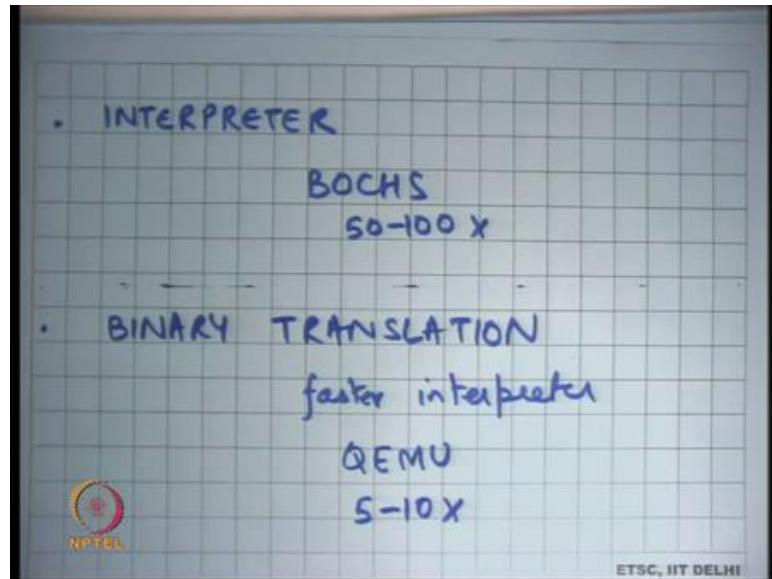
So, you can actually do migration across different machines, which would be a harder thing to do for something as amorphous as a Unix process all right. So, it can it is actually a very good fit for a distributed system where you can imagine that you have different lots of different machines, and you can take one virtual machine and just decide where to schedule this process right.

You can schedule it on this physical machine, or you can schedule it on that physical machine and so on. It also allows you to do other interesting things for example, you could run a full operating system within this container right. So, you could install let us say a Linux system, and spawn, virtual machine, and install a windows operating system inside that virtual machine.

Because, the abstraction is exactly that of the hardware, the windows operating system running inside this container well never actually know that is actually running inside a virtual machine, it will see exactly the same abstractions that would I have seen at the hardware level right.

So, there are there are lots of advantages to doing this. And, this virtualization forms the building blocks of what is also called as cloud computing and I want to discuss that more a little bit in a little bit, but let us first discuss how virtual machines are implemented it is if it comes to your question. Firstly, I need to implement them in some efficient manner; otherwise the whole thing is not practical. So, what is the simplest way to implement virtual machines, well one way to implement virtual machines is interpretation.

(Refer Slide Time: 29:41)



The interpreter based right. We just write an interpreter, which basically a (Refer Time: 29:53), which basically behaves exactly like what your hardware does and just takes one instruction executes it takes, the next instruction executes it and so on right. And, so you basically have an interpreter and you run this the code of the machine, which is executable now inside this interpreter and this the code should run right.

So, for example, a software like Bochs is a virtual machine monitor, could be called a virtual machine monitor, at least in the definition that we have so, far except that it is too slow right. It is going to take one instruction from the process, it is going to decode, it is going to see what registers it accesses, it is going to emulate those registers in memory, it is going to perform this operation and then go to the next instruction and so on right.

So, there is an interpreter loop each instruction has to internally execute 50 to a 100 more instructions and so this has an overhead of 50 to a 100x. Notice that this interpretation based approach is completely safe, the you do not need to expose raw hardware to the executable the executable sees virtual hardware. And, this virtual hardware is a virtual CPU, you could emulate a virtual device like a virtual network card and so on. And they are basically just talking to this virtual network card with the same interface that you would have had for a physical network card right.

And, the virtual network card internally is using let us say system calls exported by the virtual machine monitor to actually send packets on the physical wire ok. So, that is an

interpretation based virtual machine monitor it is really slow, but I it is all completely safe, I do not need to trust anything.

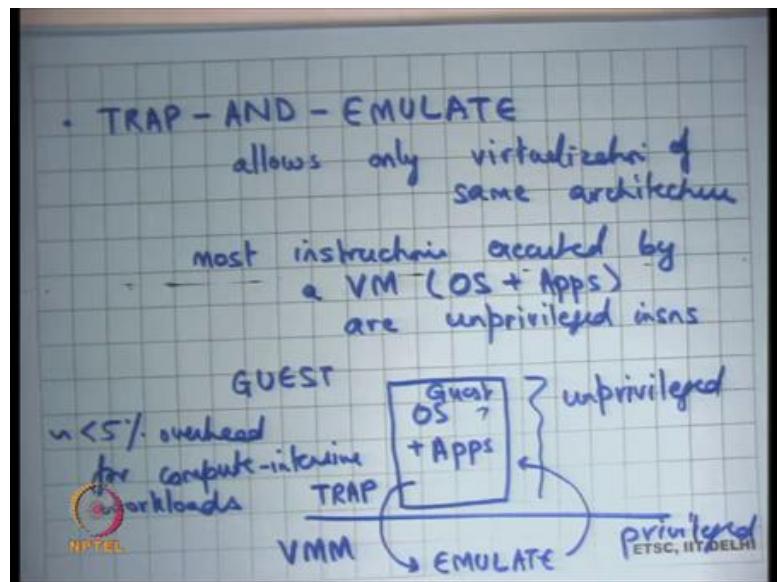
The other approach is basically, what is called binary translation, here the idea is that if there is a piece of code that gets executed a 100 times or a million times that is more likely. Then you do not need to run the interpreter loop every time you see that instruction, you can take those let us say you can take those a 100 instructions, and translate them to some safe counterpart, and then execute those 100 you know execute the translated counterpart a million times right.

So, it is a fast way of doing interpretation let us say it is a fast interpreter, a faster interpreter where basically instead of taking an instruction every time decoding it and x performing it is operations, you translate that instruction into the operations that need to be performed each time that instruction gets executed and you just jump to the translation right, that is basically.

And, an example of this kind of a system is QEMU, which is basically which you have used in your assignments they give roughly 5 to 10x overheads ok. So, it is an improvement over something like Bochs, but still not good enough right. I would not use something like QEMU to run a virtual machines for which I care about performance.

So, because they have such high overheads these software systems are not called virtual machine monitors. So, the additional requirement on a virtual machine monitor is that, the performance overhead of virtualization should be very small, it cannot be it should be in some percentage points it cannot be a 5 to 10x for example. So, the third way of and the most common way of doing virtualization is what is called trap and emulate ok.

(Refer Slide Time: 33:51)



So, what is done here is here you allow so, before I go forward let me also point out that in both these approaches interpreter and binary translation. I could have run a virtual machine for one architecture on the physical machine of another architecture nothing stops you from doing that right. I could run the virtual machine for the x86 architecture on a physical machine for power PC architecture right because I am just completely emulating the x86 architecture on power PC, it does not matter in over the underlying machine is.

Student: So, if the monitor would be modified accordingly.

Well the monitor you know will be modified accordingly in the sense that let us say if you are writing it in a higher level language like C, then it should be compiled for power PC that is all right. If, the underlying machine is power PC then you will compile it for power PC that is all otherwise the monitor's remains the same, but something like a trap and emulate allows only virtualization of same architecture.

So, which basically means that you can only run the virtual machine of the same architecture as the underlying physical architecture ok. And, the way it works is the key observation is that most instructions executed by VM and VM basically I mean the operating system that is running, and the applications are unprivileged instructions.

So, a huge majority 99.9 percent or more of the instructions that get executed inside a virtual machine, instructions that are very regular in nature like move, add, subtract, load store etcetera right. So, these are very regular instructions, these instructions would have had the same effect whether they are run in the user space or whether they have they run in the kernel space.

And, and so, these instructions the emulation of these instructions can, or the translation of these instructions can be made identity. So, if the guests are so, if the process which I am also going to call the guest right. So, the guest executes an instruction called an unprivileged instruction, I can just execute the same unprivileged instruction in the host and get the same behavior right.

And, so what you do is you take the operating system and the applications. So, the guest operating system and applications and run it in unprivileged mode right and the virtual machine monitor runs in privileged mode. Because most of the instructions are unprivileged in nature when they execute, they just execute as though they are running natively, if they execute any privileged instruction. For example, if they try to access CR3 right or if they try to execute in instruction or an out instruction to access the device you trap ok.

Just like and then you emulate just like you were doing in Bochs and QEMU you emulate inside the VM and then you return. So, what you do is you take. So, consider this disk image of a virtual machine as your executable right, that is the; that is the disk image, that is basically that basically lives on your hard disk for a physical machine, that is a hard disk (Refer Time: 38:05) the hard disk is basically the executable.

And, so now this hard disk in the virtual world is going to live in a file that is called we are going to call your virtual disk, and I am going to run this virtual disk inside this container called the virtual machine. And, this virtual disk is going to start running some of these instructions one after another, what you are going to do is you are going to run these instructions all these instructions in unprivileged mode. Even though these instructions were meant to some of these instructions, were meant to be run in privileged mode.

For example, when you boot a machine the first few instructions the assumes there is a privileged mode, anytime you transition to the kernel you assume that these instructions

are running the privileged mode, but the change you make as you run all these instructions in the unprivileged mode. Most of these instructions execute without any problem, because most of these instructions are unprivileged instructions. Any instruction that is a privileged instruction that must have been an instruction inside the guest who is kernel will cause a trap right.

So, you rely on some hardware properties that any which basically say that if you have tried to execute a privilege instruction in an unprivileged mode you always get a trap all right. So, it causes a trap and you handle the trap by emulating that instruction in software, just like you had done for Bochs or QEMU and then you return back to your guest right.

So, this kind of a virtualization is called trap and emulate virtualization and significantly faster, then either interpretation or binary translation and you will for something which is compute intensive you will have less than 5 percent overhead for compute intensive applications ok.

On the other hand, something that is very IO intensive, it is very likely that you are going to be executing lots of privilege instructions. If the guest is exec trying to access lots of you know virtual devices, then it is probably executing lots of in and out instructions or something that is privileged. And, anything that is privileged is going to cause a trap, it is going to execute some emulation logic inside the virtual machine monitor and then return back and so, there is extra overhead of doing this trap and emulation is going to become visible on IO intensive application.

So, on in IO intensive application, you can have overheads of up to let us say 2 to 3x or 4x, you know you have got go back get back to the QEMU kind of performance for IO intensive application. And, this is just a basic trap and emulate style virtualization there are many optimizations that can be done over it, and today you know with hardware support, and all that you can do virtual machine monitor virtual machines at almost negligible overhead.

So, what have we achieved we basically have a process abstraction that looks very much like the hardware abstraction and it runs with almost 0 percent overhead right? That was you know that is the whole idea of a process abstraction in the first place right. The

process should not be running too much lower than when you are running it over kernel as opposed to not running it over the kernel.

Is the VMM running on top of the kernel well consider VMM as a part of the kernels. So, VMM is yet another subsystem inside the kernel just like fs, virtual memory, drivers, there is yet another type system inside the kernel called the virtual machine monitor, great. It is a relatively modern subsystem, because you know virtualization is a relatively modern thing, it was very popular in the early days of computing in the 1960s, when machines used to be very large like the IBM mainframes and used to be really expensive.

So, the only way to actually make full use of these machines was basically lots of people sharing these machines and at that time virtual machines was a very popular concept and lots of research was done on you know using implementing fast virtual machines.

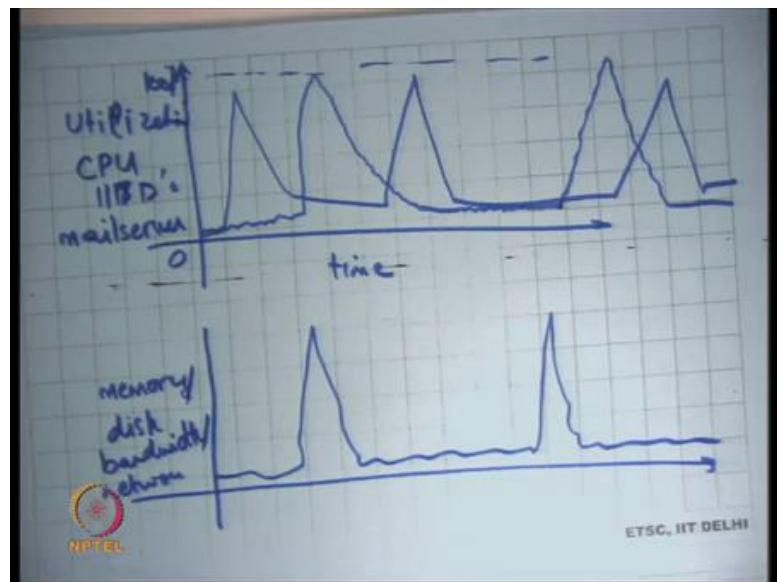
Over the years computers became smaller, they became personal, things did not have to be shared, computers did not have to be shared and so, virtualized it the technology of virtualization was lost. What do I mean by technology of virtualization was lost the hardware interfaces were not virtualization compatible?

For example, recall that one of the requirements to implement this trap and emulate style virtualization is that if I run a privilege instruction in an unprivileged mode it must trap right. So, but the hardware designers just did not care about it, because they thought these are personal computers. And, similarly the software stacks were not tuned for virtualization etcetera.

But, it was rediscovered because now today even though our hardware is relatively much cheaper the cost of actually managing this hardware, which includes power, store, power space, people who know how to manage these things, software maintenance up gradation, security viruses etcetera. All these problems basically make you know have moved us back to a and you know made the case for again a centralized computing infrastructure that lots of people are sharing and that is what we are calling cloud computing today right.

And, so one of the building blocks of cloud computing is virtualization. So, let us see why virtualization is helpful or why you know why cloud computing is a useful thing.

(Refer Slide Time: 43:47)



If, you look at the utilization levels. So, let us say this was time and let us say this was utilization right. So, what do I mean let me take one any one machine, let us say let us take the example of a web server, let us take the example of the web server of IIT Delhi right?

So, if I look at the CPU utilization of IIT Delhi's web server, a probably see in a let say this is 100 percent and this is 0, then it will probably we somewhere like this. Sometimes, there will be a surge some results are getting announced and then there is like this and then there is a surge again let us say etcetera right.

So, this will be the typical curve for CPU utilization of a server, same things for let say memory or disk, band width, or network right. So, usually we will get some kind of behavior, where on average the utilization levels of the machine are very low, but sometimes there is a very high surge in the demand, and you get very relatively higher utilization there was. And, so what typically do is we provision for the maximum we tried to provision for the maximum. So, the when there is a very high load, I do not see problems.

But, most of the times this maximum is actually getting wasted it may not it is getting wasted because I am probably consuming lot of power is still, I am consuming less power at 10 percent utilization and then when what I consuming 100 percent, but still I am consuming power and I am consuming the equal amount of the space definitely and

so on right. What would have been better was if I had if I was able to run multiple machines on the same physical machine.

So, let say I have you know I have the IITD's web server and let say I have mail sever, which is let say my departments mail server the computer science, department mail server. And let say it has behavior like this, then these can easily be multiplexed one the same physical machine without seeing performance degradation on either application or yet having better utilization over for my physical machine over all right.

So, virtual machines allow this ok. And, they allow this without having to worry about software compatibility, you may say what is a big deal why do not I just have one Linux machine and run the mail server as one process, my departments mail server is one process, and you are your institutes web server as another process, and let them you know co-exist, and it is going to do exactly what you wanted to do.

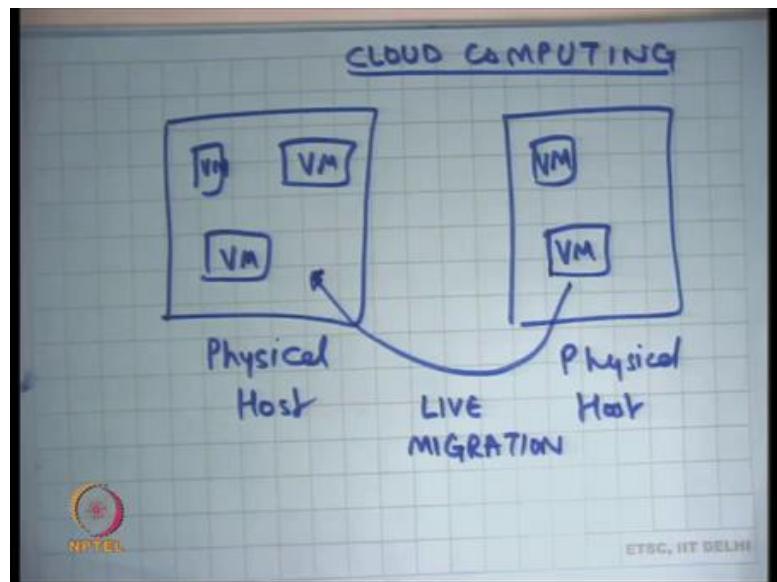
But very difficult to because the abstraction are so, fragile it is very difficult to ensure that the departments web server can be run in to one the same operating system as the department institutes mail server without causing any problems with each other and without having any security implications right. So, this department web server may be less critical than the institutes web server and so, you may want that you know that it should be completely isolated from each other.

It something may not be that carefully written other thing may be carefully written. And, because the abstraction of the Unix abstraction let say, are very fragile you know for example, the configuration files etcetera, you know these processes will depend on the configuration files this slash etc folder etcetera.

It becomes a lot of it becomes a mess after less time to manage lots of different processes running on a single operating systems. On the other hand, the virtual machine abstraction is very clean, it is the hardware abstraction. So, I can just take one hardware abstraction and run it and I can I cannot have to I do not want to worry about security problems or interference problems between these two virtual machines.

So, what this allows me to do is basically let say, I have this is a physical machine let me call it a physical host and these are virtual machines, VM, VM, VM.

(Refer Slide Time: 48:07)



This is another host this is VM right. So, I am showing you cloud computing scenarios what is called cloud computing in the modern world. So, this is what a cloud computing environment will look like, you will have multiple physical hosts and you would be running typically multiple virtual machines within a physical host. Typically, you would be you could depending on the utilization levels of each VM, you know you could pack multiple VMs on the single host it is very common to see tens of VMs packed on a single physical host. Because, the utilization levels of each of these VMs is very low right.

And, so you could be running these things together the nice thing is because is abstractions are so, tight I can at any time decide to move this virtual machine from here to here right. So, this capability is what is called live migration, this can be done in a way such that the virtual machine has near 0 down time right. So, it appears that the virtual machine never switched off right.

So, there is a you know one way to show this is basically this is interesting demo where let say you are watching movie on a virtual machine, a live migration happens which means the virtual machine shifts from one physical host to another. There is some amount of copying that is going on for all the memory state etcetera, it is all happening in the background and at some point the entire virtual machine shifts from one host to the other and you do not want to see of blip in your movie right.

So, that is basically that is what I mean by a near 0 down time live migration. So, that is the very wave power capability what allows me to do is it basically allows me to dynamically schedule my work across my physical host, at will and that is one of the main strength of cloud computing. Let say I have you know let say these 3 VMs become very highly utilized certainly.

So, I can choose to move this VM from here to here right, it is all let say it is in the middle of the night and none of these VMs are actually using any resources at all. So, I can pack all these VMs on a single physical host and actually switch off this particular physical host and save power right. Also, I can do fault tolerance which basically means now your virtual machine state can be encapsulated as a file.

So, running virtual machine can be snapshoted and its state can be preserved, its live state can be preserved as a file, and it can be loaded from where it was at will to basically get exactly what it was when it was snapshoted right. That allows me to do fault tolerance; fault tolerance basically means, if there was an error in either hardware error, or network error, or power failure it can revert to this snapshot of a virtual machine and basically get the same kind of behavior.

So, cloud computing has its advantages basically, they have talked about for the more interesting thing from operating system standpoint is that it brings open all the issues that we thought were dead for a long time. For example, scheduling now becomes a very important issue right. If, you are running a cloud computing facility if you run a data centre and becoming very important that you manage the data centre with maximum utilization and the maximum utilization depends on what scheduling you will get them you using right.

And, like the personal computer world were generally the hardware resources are plenty in plenty, in cloud computing environment you basically sharing these resources across lots of people and you have minimize trying to minimize your costs. So, resources are never plenty you are basically always trying to optimize your resources in some sense all right ok.

So, with that we finish this course.

**THIS BOOK IS
NOT FOR SALE
NOR COMMERCIAL USE**



(044) 2257 5905/08



nptel.ac.in



swayam.gov.in