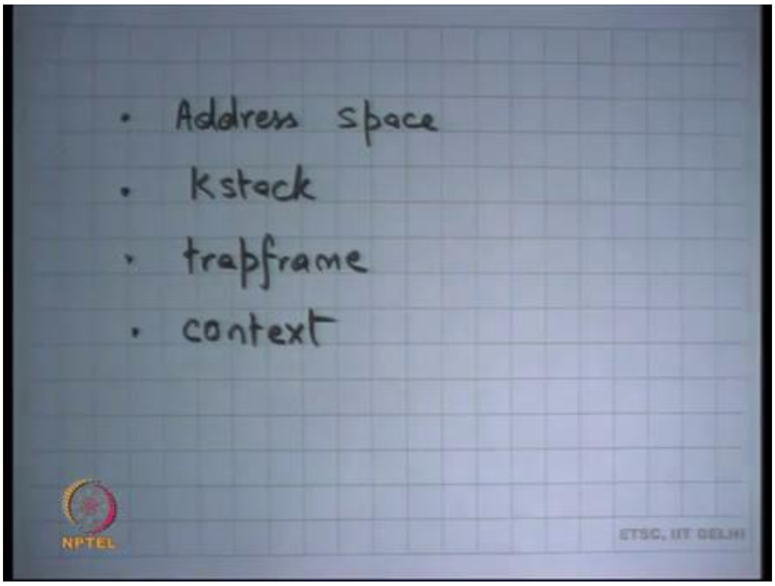
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

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)



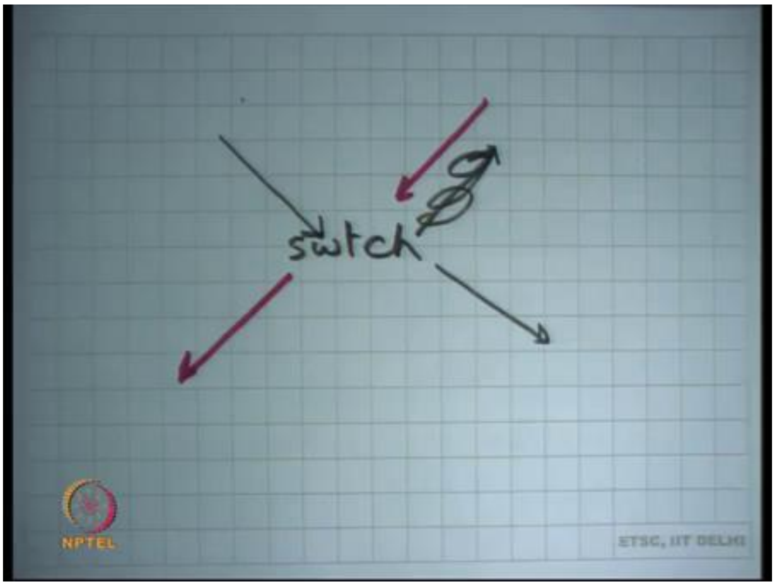
- Address space
- kstack
- trapframe
- context

NPTEL
ETSC, IIT DELHI

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 calls 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.

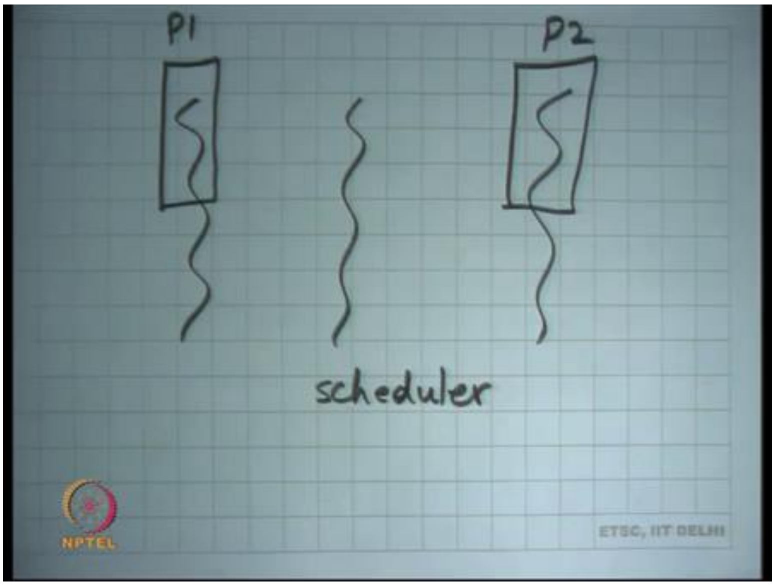
(Refer Slide Time 02:05)



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 it is 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 it 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 want 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 in 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?

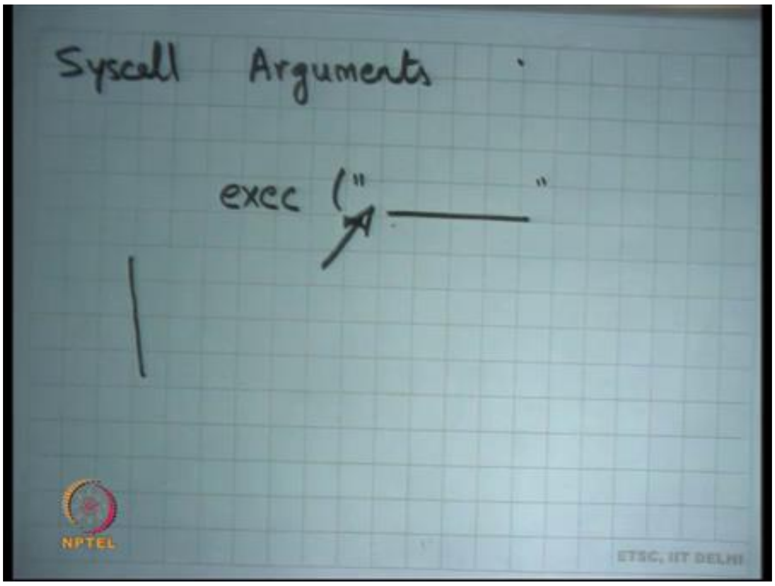
Student: No.

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.

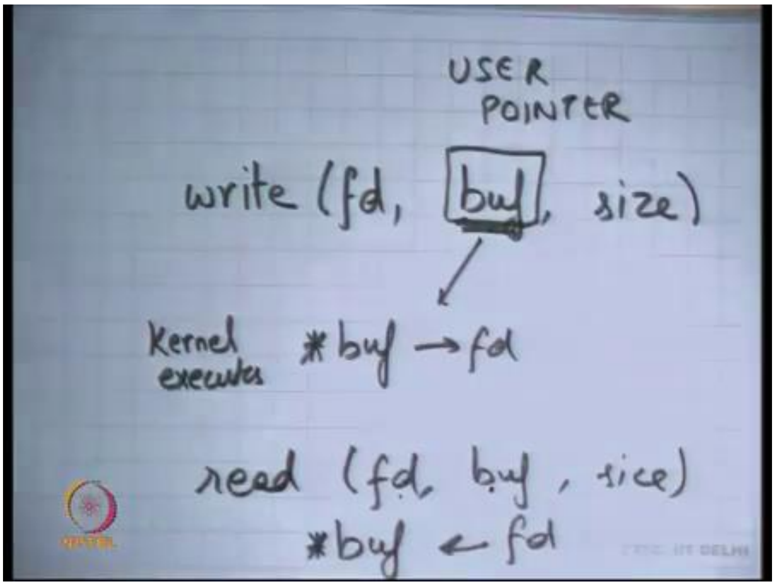
And, what we said was let us say I have a system call called exec and it takes a string, right. Now, a string can be of arbitrary length. So, clearly a string cannot be passed into registers or even you know on passing the string on the stack may not be the right thing, because you know stack it is you 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)



USER
POINTER

write (fd, buf, size)

Kernel executes *buf → fd

read (fd, buf, size)
*buf ← fd

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.

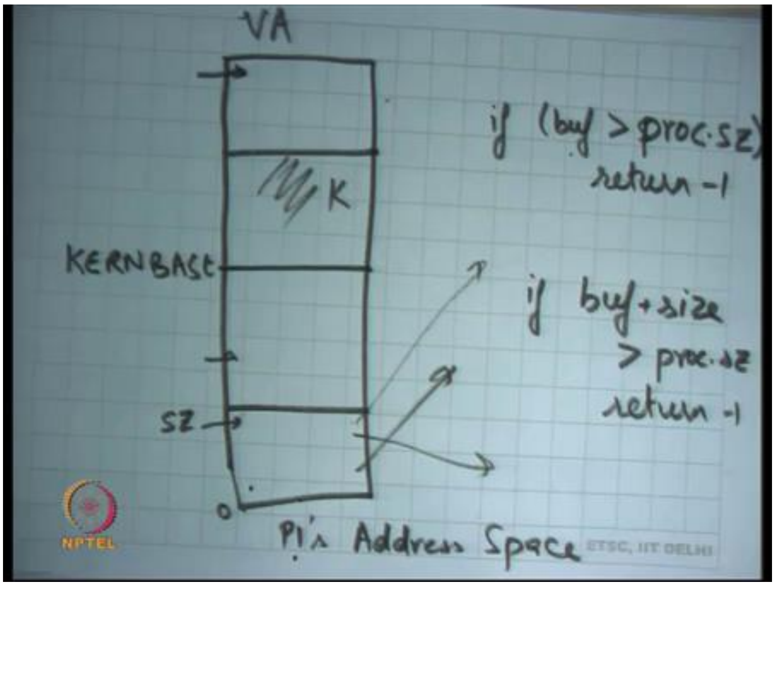
Right so, it should be an address that is actually mapped in my address space. It because if it is not mapped then what I can do, what the user can do is ask the kernel to deference a location which will cause a page fault, right. And, the kernel is, may not be capable of handling page faults while it is executing, right. It 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 faultsm, 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 buff 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)



```
if (buf > proc.sz)
    return -1

if buf+size
    > proc.sz
    return -1
```

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.

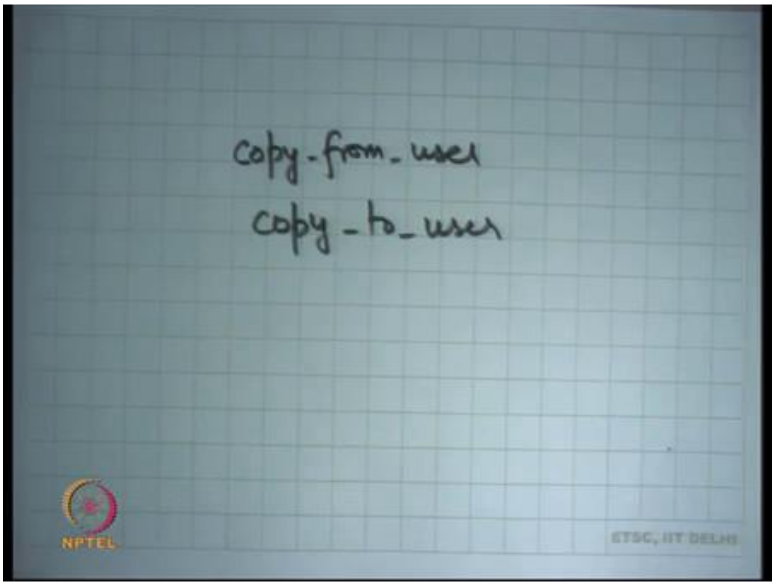
(Refer Slide Time 19:57)

fetch int

fetchstr

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)



copy-from-user
copy-to-user

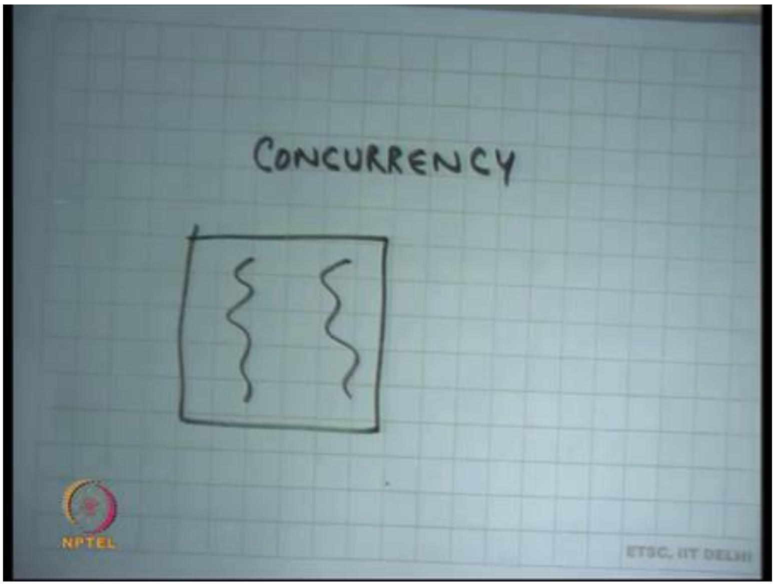
Similarly, on Linux, where these functions called copy from user and copy to user, right; once again, these are special functions designed to handle user pointers, right. If you happen to miss; so, for example, you know kernels code is relatively large and it is possible and multiple people are contributing 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.

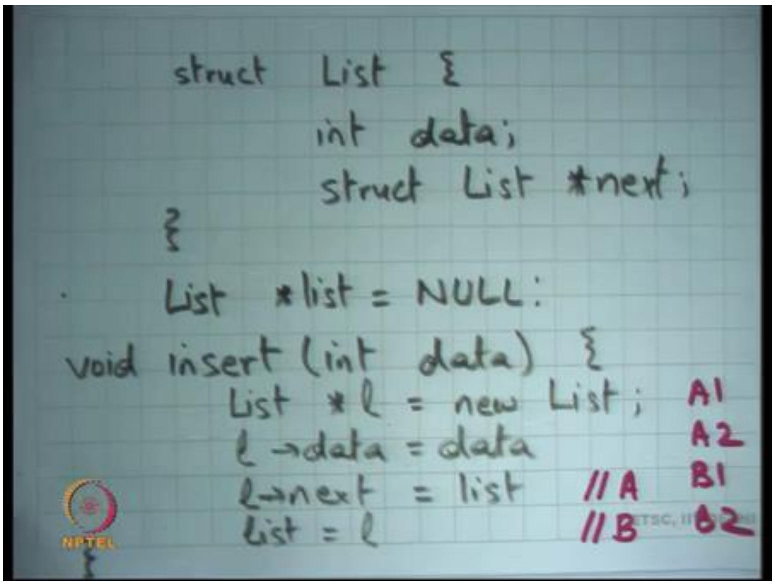
(Refer Slide Time 22:38)



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)



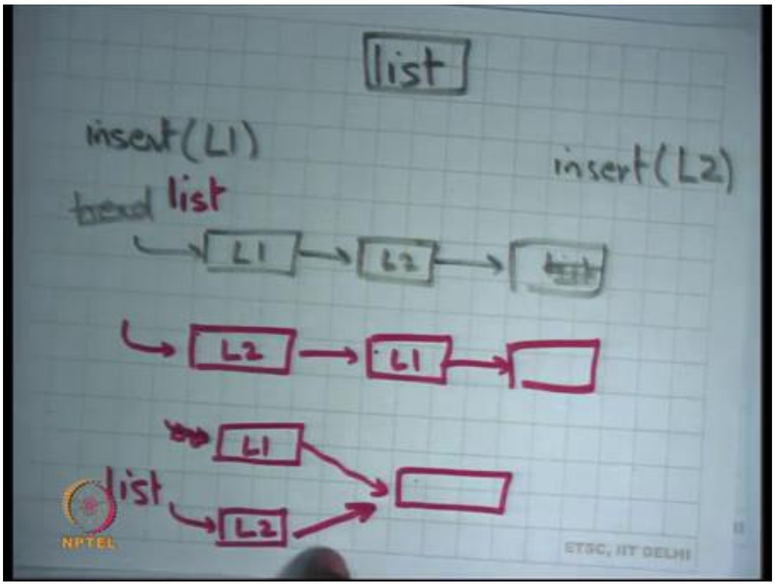
```
struct List {
        int data;
        struct List *next;
}
List *list = NULL:
void insert (int data) {
        List *l = new List;          A1
        l->data = data               A2
        l->next = list       //A     B1
        list = l             //B     B2
}
```

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.

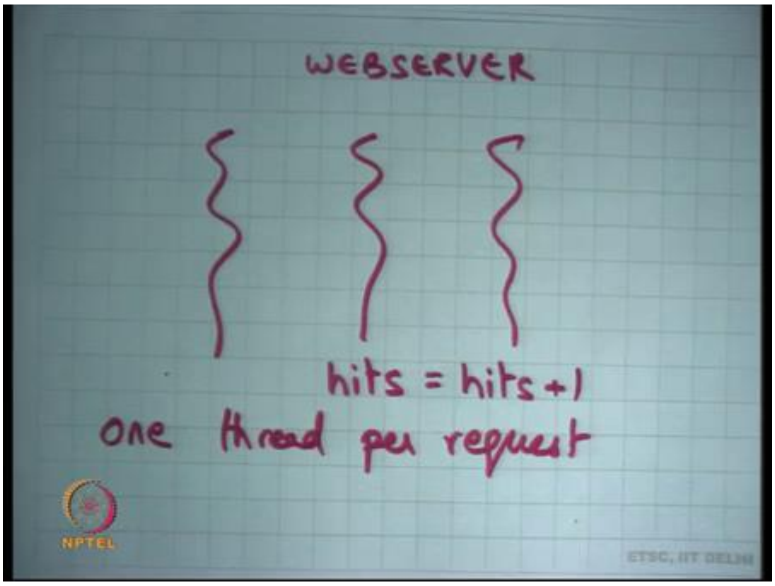
This is just synchronization in your own mind going on between one program vi and another program gcc, right. If you do not wait for the same message to appear and you try gcc apriory then bad things are going to happen, right. And you cannot even predict what kind of bad things can happen right which 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)

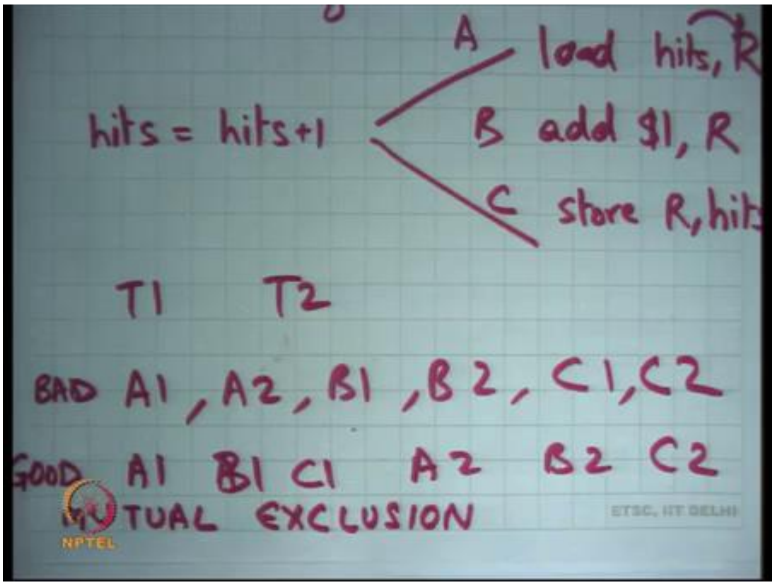


Let me take another example. Let us say I am a web server and I have multiple threads in the web server, right. You are running a website and your web server and what you have done is you have said that each thread is going to handle one request. So, one thread per request, right; so, whenever I type 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)



hits = hits+1

A load hits, R

B add $1, R

C store R, hits

T1 T2

BAD A1, A2, B1, B2, C1, C2

GOOD A1 B1 C1 A2 B2 C2

MUTUAL EXCLUSION

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?

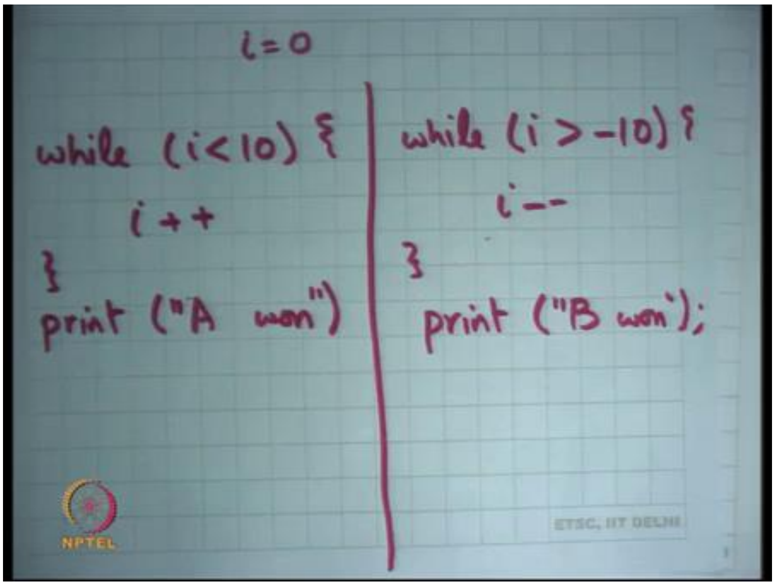
Student: Hits plus 1.

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)



```
              i=0

while (i<10) {          while (i > -10) {
      i++                     i--
}                       }
print ("A  won")        print ("B won");
```

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)

AVOID BAD INTERLEAVINGS

LOCKS

NPTEL
ETSC, IIT DELHI

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.