High Performance Computing

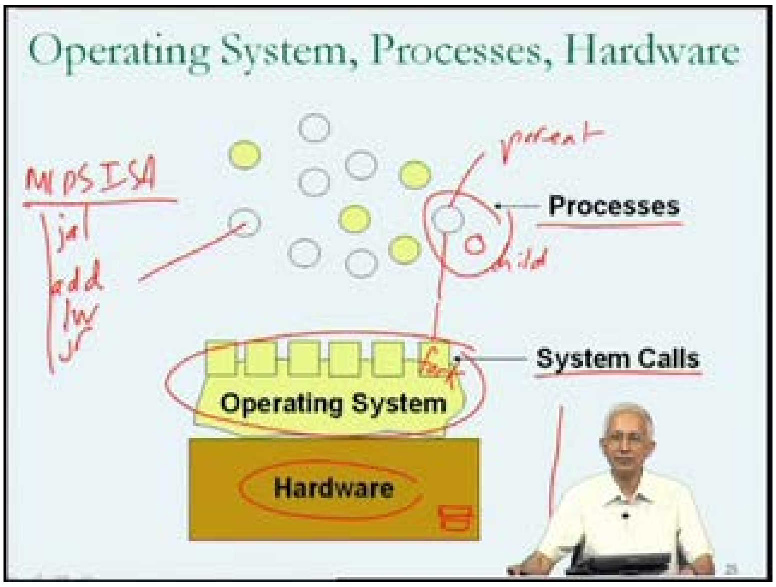
Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 03
Lecture No. # 12

Welcome to lecture number 12 of our course on, High Performance Computing. In the previous lecture, we have seen or started to look at the software side of the picture, other than the programs that we write, there is a fair amount of important and interesting software, which helps to determine what happens when our program execute. In the slide which we are using to represent the computer organizations software side of things, you remember that the hardware is the base and I will direct you to look at that slide.

So, the hardware is the base on which the software organization of the system is built. That is why at the bottom of the slide on software organization of the computer system, we have box label hardware and that box contains, as I said, the ALU, the processor, the I O devices, bus and so on. Above this, we understand there are going to be various software components, because it is within the hardware box and only within the hardware box that they are going to be hardware components, the rest of the organization of a computer system is going to be through the software components of the computer system.

(Refer Slide Time: 01:32)



So, in the diagram on computer organization software that we were using in the previous lecture and which I would now request to come up on the screen. We have the hardware box at the bottom of the screen and above this; we have various software components of the activity on the computer system.

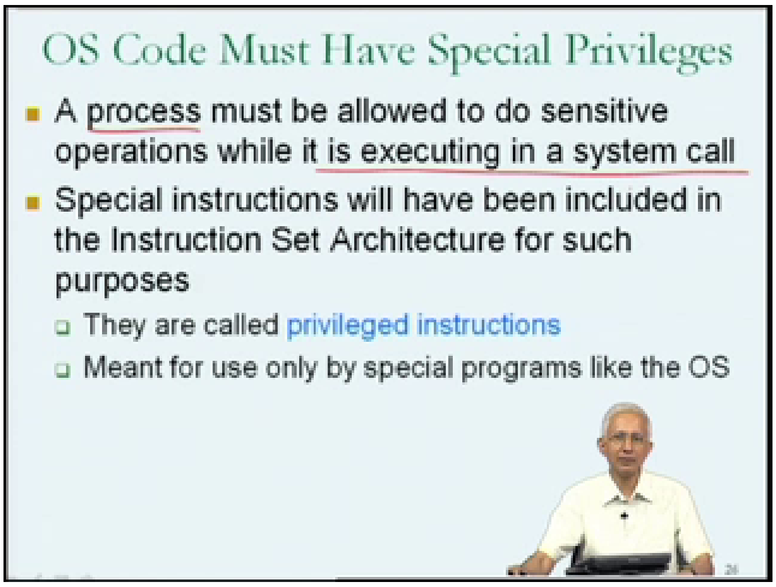
So, you will notice that we have various entities, which are either in white or in yellow and it is only the hardware that we have been talking about in prior lectures. So, we are now using the term, Process to refer to programs in execution and there is a very key operating system; the key component of the software of the computer system, which is represented closes to the hardware in this particular diagram.

Now, the different processes or programs in execution when they require functionality of from the operating system will do so by calling function like entities know as, System Calls, which are inter phases into the operating system, and last time we saw some interesting system calls. For example, there is one of the system calls known as fork. If a process calls fork, then the effect is that on its behalf, the operating system does something through which a new process comes into existence, and the original process we hence forth refer to as, Parent Process and the new process, we refer to as a, Child Process and in fact, both the parent and the child process execute the same program. They are two processes which are now associated with the execution of a single program.

Some of the processes that we had shown were clearly operating system related, because they are shown in yellow which is the operating system color on this slide. Now, in the end of the previous lecture, we were discussing how the yellow parts of this computer organization software slide, such as the parts over here, must be capable of doing very privileged operations on the hardware. Now, the ordinary white processes will use all the instructions that we saw like add, load word, jump, register, jump and link and so on. But, as far as, the operating system is concerned, there may be new to use some very privileged registers or some very privileged other capabilities that are present in the hardware and not made available to the user programs.

So, what we saw in the MIPS one I instruction set architecture was clearly just a subset of the actual MIPS one capabilities. The MIPS one processor is capable of executing all the instruction that we saw, but, in addition some additional instructions that would be included for the use of very privilege program such as the operating system.

(Refer Slide Time: 04:07)



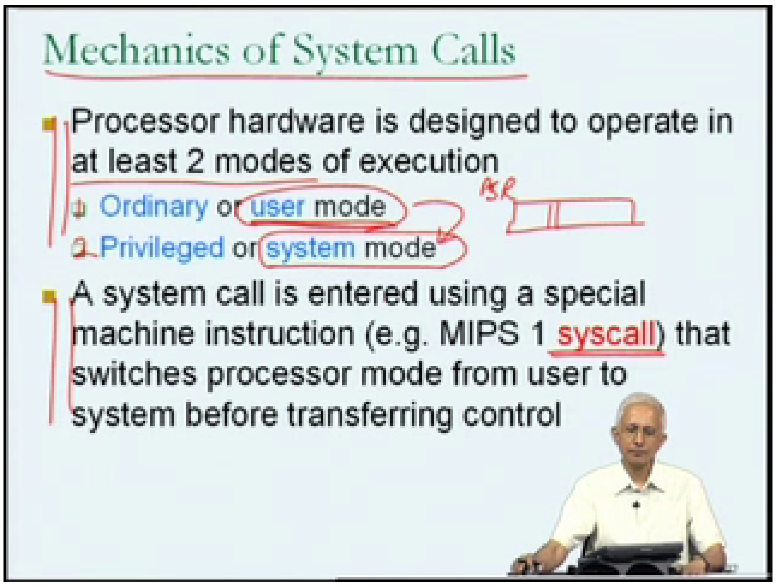
OS Code Must Have Special Privileges

- A process must be allowed to do sensitive operations while it is executing in a system call
- Special instructions will have been included in the Instruction Set Architecture for such purposes
  - They are called privileged instructions
  - Meant for use only by special programs like the OS

So, basically the complication that we have now is, we understand that such privilege instructions might be included, in the instruction set, for specific use by the operating system, and special programs of that kind, but when a program that you or I write, runs as a process, and that process makes a system call, which is a definite possibility, as we saw, I can write the program, which calls fork or exact or s break or open, close, read, write, so many system calls that are very useful to programs that you or I might write.

When my program is executing inside a system call, it is executing in the yellow part of that diagram, and it is executing a piece of code, which might have to make use of privileged instructions, and that is not something that I should be able to do in a program that I write. If I can write a program, which can execute privilege instructions then I could also take the privileges that should be available only to the operating system.

(Refer Slide Time: 05:10)



Mechanics of System Calls

- Processor hardware is designed to operate in at least 2 modes of execution
  1. Ordinary or user mode
  2. Privileged or system mode
- A system call is entered using a special machine instruction (e.g. MIPS 1 syscall) that switches processor mode from user to system before transferring control

So, this is the subtlety that has to be addressed very carefully by the hardware. So, we need to understand what happens when a system call takes place. Clearly, it is not just going to be like a function call. In a function call, there is the need to save return address and all that complication that we saw, but ultimately there was just a control transfer. Whereas, in the case of a system call, there is very clearly the need for a control transfer, but also the need for some kind of a privilege elevation, a change in the status of the process which is making that control transfer.

So, in order to achieve this, what is done is that processor designers design the hardware of the processor, so that it can operate in either of multiple modes of execution, at the very least, would need at least two modes of execution. And as you would imagine, one of the modes is going to be available, and it is going to be an ordinary mode, which is the kind of mode in which instructions of the kind that program of yours or programs of mine might require. The ordinary instructions that we have seen of the MIPS one instruction set.

So, the hardware will be designed, so that it is either executing in this ordinary or user mode or in a more privileged mode call the system mode. So, at any given point in time the processor is executing instructions and it is an either user mode or the system mode. Therefore, if it is in the user mode, it can execute the ordinary MIPS one instruction, but not the privilege instructions, whereas, if it is in the system mode, it is capable of executing the privilege instructions.

What will happen if I write a program and I include a privileged instruction in the program? Now, when I write a program and it runs as a process, the process will be running in user mode until I try to make a system call. So, if my process is running in user mode and it tries to execute privilege instruction, the hardware will notice that a processor is currently in user mode, and it will give an error violation, indicating that an attempt was made to execute privilege instruction and I will find out that my program is not allowed to do so.

Whereas, if the program is executing in the system, is executing in the operating system or executing inside the system call, it will be in the system mode, and if it executes the privilege instruction while in system mode, the hardware will know that this is a allowed execution of a privilege instruction. How does the hardware know whether it is in user mode or system mode? You remember that there are special purpose registers, inside the hardware. We talked about processor status register. So, it is conceivable that one of the bits of the processor status register indicates what mode the processor is currently executing in. Now, I have indicated that.

So, the current mode of a processor might be indicated in processor status register. Remember, that the processor status register, like the program counter or the instruction register are special purpose registers within the CPU, used by the control hardware. In this bullet (Refer Slide Time: 08:07), I have suggested that they need to be, hardware has to be designed with at least two modes of execution, a user mode and a system mode. Processor could be designed to have three or four modes of execution, in which there are some refinement of the different kinds of functionality that might be available in the privilege mode.

But, for our purposes we will just assume that there are precisely two modes. The ordinary mode in which my program or your program runs and the privilege mode, which is the mode, in which the operating system programs or operating system code executes. Therefore, as far as the mechanics of a system call is concerned, we understand that if my program executes, if I write a program which makes a system call, then in order to execute in the system call, my program has to change mode from user mode to system mode. And there is no other way that my program can execute the system call, because the system call code basically contains privileged instructions.

How can this mode change a change of mode take place? The answer is this will have to be done explicitly by the execution of an instruction and thus we saw the MIPS one instruction set had an instruction called sys call, which we can now interpret as system call. If my program executes the sys call instruction, then essentially its switches mode from processor mode to user mode, before transferring control to the appropriate system call.

So, the sys call instruction can be used to switch modes, in order to transfer control to one of the specific system calls, and not arbitrarily just changing mode from user to system. So, it is a control mechanism for changing from user mode to system mode for the sole purpose of executing in the system calls.

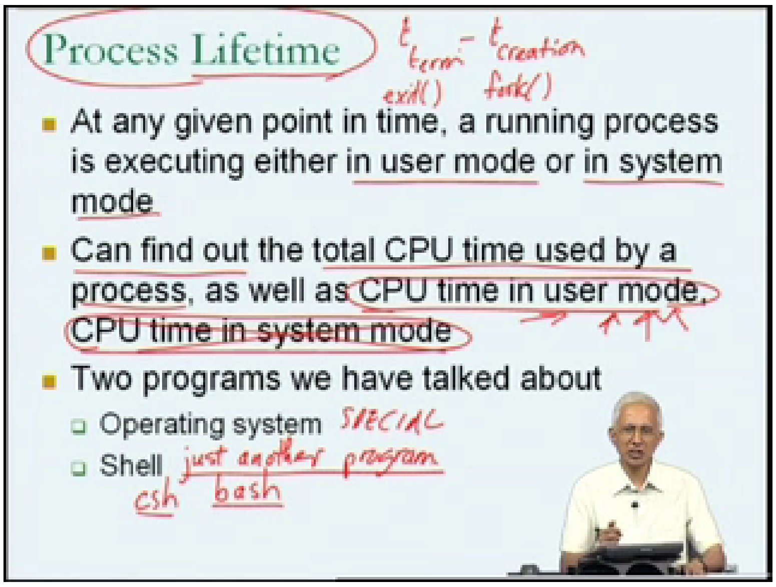
You will notice that this is an interesting instruction, sys call, because it is an instruction the does transfer of control. It clearly must save return address, because ultimately after the system call is executed, control must be transferred to the instruction after the fork in my program, after the system call in my program. But, more curious than that is this call instruction itself cannot be a privileged instruction. It has to be an ordinary instruction, because the sys call instruction has to be permissible in ordinary user programs.

So, this is an instruction, which is an ordinary instruction, which has a side effect of changing from user mode to privilege mode. It does other things as well, such as saving return address and transfer of control to the appropriate system call, the beginning of the appropriate system call and possibly some of the other complications that we saw as for as function call is concerned. So, this is essentially what happens in a system call. You notice that there is a need for hardware support as far as system call is concerned. So, it is conceivable that they are processors that exist today on which Unix cannot be ported, just because they do not have features of this kind.

It may be possible for somebody to attempt a port of the Unix or Linux operating system to such a processor, but it may end up being either extraordinarily difficult, because it is a lack of hardware support or it may end of being not possible at all.

So, in the design of processors, the architect must take into account certain requirements of the operating system and we see that they are linkage between the operating system and the hardware is possibly closer than we thought. In designing the hardware, in designing the instruction set, the compute architect does not just take into account the requirements of ordinary programs, must also take into account they very specific and privilege requirements of the operating system.

(Refer Slide Time: 11:47)



Process Lifetime

$t_{term} - t_{creation}$

exit() fork()

- At any given point in time, a running process is executing either in user mode or in system mode
- Can find out the total CPU time used by a process, as well as CPU time in user mode, CPU time in system mode
- Two programs we have talked about
  - Operating system SPECIAL
  - Shell just another program
    csh bash

Now, moving right along, I am going to refer to the Process Lifetime. I have used a word, lifetime before. We talked about lifetime of a piece of data, as being the laps time or the time interval between the time that it seize to exist and the time into time instant that it seize to exist and the previous time instant when it came into existence. Some pieces of data have life times which are the life time of the program in which they come into existence. Other variables, other pieces of data have life time, which is related to a function call, others have life times which are related to explicit points in the program, for example, time interval between malloc and free.

We can also refer to lifetime of a process, by which we would mean the time between the termination of the process and the creation of the process. This is what we would refer to as a lifetime of a process; the time between the termination of the process and its creation (Refer Slide Time: 12:51). We know that the process is created by a call to fork and process terminates due to call gracefully on a call to exit, and therefore, the lifetime of a process could be quite clearly calculated, quite clearly measured.

Now, while a process is in execution, it might be interesting to keep track of how much time it has been in execution, in other words, the amount of time associated with that process at that instant. So, it turns out that in UNIX systems, as we have seen, at any given point in time, a running process is executing either in user mode or in system mode. It will be executing in user mode if it is executing the instructions of your program. It will be executing in system mode, for example, if it is executing a system call, as requested by your program. Therefore, if I talk about the lifetime of a process as its time interval, some components of the time interval are going to be spent executing in user mode, and some components of that life time are going to be spent executing in new system mode.

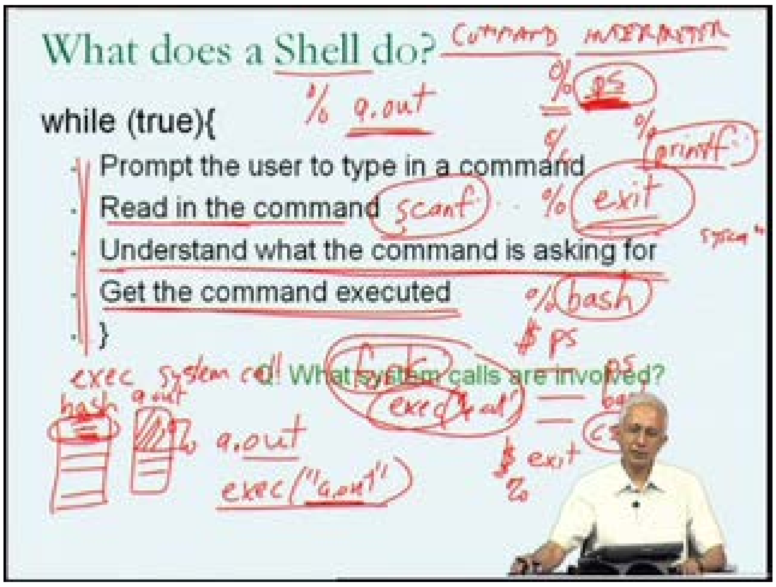
Therefore, I could actually talk about the time in user mode or the CPU time spent by my program in user mode, as well as the CPU time spent my program in system mode, and possibly the sum of these two, would give me some idea about the total amount of CPU (( )) time that my program in execution took, and these might be interesting values to know, because we talked about how the objective of taking this course was to try to improve our programs, and one aspect in which we may want to improve our program is by reducing the amount of time that it takes to execute.

Therefore, I would be talking about trying to reduce the total CPU time used by the process, which is my program in execution, and while it might be possible for me to control the amount of time in user mode, it might not be possible for me to control the amount of time in system mode, without reducing the number of system calls that my program makes.

So, if we make sure that the number of system calls that our programs make are as required for the functional, for achieving the functional requirements of the program, then the second aspect is outside of our control. All that we can do is to rewrite our program, to improve the amount of CPU times spent by our program in user mode. Therefore, this is an important distinction and as I have indicated over here, on Unix or Unix like systems, there are facilities provided through which you can actually find out the amount of time, the amount of time CPU time, spent by a process while the process is running. In other words, one could make calls to those facilities from within the program and then printout the values of the amount of time, total the CPU time spent or the total amount of CPU time spent in user mode, total amount of CPU time spent in system mode, to get some idea of what your program is doing in terms of the amount of time it has spent.

Now, let us go back. In talking about computer organization, the software side of things, I have talked about two programs. One is the operating system and strange enough, the other was shell and I had suggested that the operating system is very special, where as when I talked about the shell; I said it is just another program. We have seen two examples of shell. I talked about CSH and BASH and you could if you have access to a Linux or Unix System, you could read more about either these shells, but just to convince you that a shell is a just another program, we will next try to understand what a shell does, in fact write pseudo code for a shell, a very simple shell.

(Refer Slide Time: 16:40)



What does a Shell do?

```
while (true){
    Prompt the user to type in a command
    Read in the command
    Understand what the command is asking for
    Get the command executed
}
```

What system calls are involved?

So, what does a shell do? Now, you will recall that shell is the word that is used for command interpreter. So, a shell takes a command from the user and it interprets or causes to get achieved that command. We had examples where, in response to the request from the shell for a command, if I typed in p s, which is the command that can be used to gets status information about the different processes is running on a system, then the shell would take the information from me, and it would cause the p s program to execute and I would see the output from the p s program on the screen.

Basically, that is what a shell does. So, we see that what a shell does is, it prompts the user for input then after the input is typed, it is prompt user for a command and after the command is been typed in, it obviously reads in the command, understands what the command is suppose to do, and then achieve the command and after doing that it prompts the user for another input.

So, just putting this down, one thing which shell has to do is prompt the user to type in a command and that is actually quite easy for us to do. You know that you could write a program which could do this, all that you have to do is to print the character percent to standard output (( )), this standard output is related to the console.

So, this is actually very easy to do, but the first thing that the shell has to do is print the shell prompt on the screen. That is what I mean by prompt in the user to type in a command. After this, it has to wait for the user to type something in, and it has to read in the command. What is involved in reading in the command? The command would have to read into a variable or in array of some array of characters. An array of characters would have to be declared, command would have to be read into that array of characters, using any of the standard I/O functions that you might be familiar with, such as scanner, for, something like that.

After this the shell would have to understand what this command is asking for. For example, in the case of p s, understand that this is the command, which is requesting that the p s program be executed. In after having done this, the shell would get the command executed. So, the next thing that it has to do to get the command executed, and having done that, it repeats this process. So, basically the shell is this infinite loop, which prompts user to type in a command, reads in the command, understands what the command is asking for and gets the command executed.

Now, we do need to understand little bit more about what is involved in all these things, all these steps, but let me just zero in on two aspects of what the shell is doing and one is understanding in what the command is asking for. Now, this is not a trivial step in the operation of a shell, because they are actually many different kinds of operations that one could type in response to a shell prompt. One thing that one could do as we saw in the case of executing a program is to type the name of an executable program file.

You will recall that if I had a program in C called program dot c and I compiled it using gcc program dot c, then I get as output file called a dot out, the default name, which is an executable or object file or a machine language equivalent of the c program. It is a file which contains the machine instructions, and various other things, which are equivalent to the program dot c that I wrote and I can cause this to be executed.

So, in response to the shell prompt, if I type the name of an executable file, then that is our request to the shell to actually run that executable file, but there are other things that I could type in response to the shell prompt. For example, the shell may have been written so that the shell has certain specific commands of its own, which do not correspond to program files. For example, most shells will have their own exit command and the purpose of the exit command is for the shell to terminate.

Therefore, exit is not the name of a program, it is not the name of a program file, but exit is the name, it is just one of the commands that the shell can inputs and understands that this is the request for the shell itself to terminate. And you ask yourself what sense does it make to ask for a shell to terminate? Let me put it to you this way. Just suppose that I am running on a computer system or UNIX system, in which by default the shell that I am given is CSH. So, by default when I give the shell prompt, I am getting a CSH shell prompt. Now, what if I want to actually change to bash shell? For the session that is going on right now, I know that bash is the name of just another program from the previous slide.

So, I know that somewhere out there is file called bash, which is an executable a dot out equivalent. It is not called a dot out. Its name has been change to bash, but it is of the same format. It is an executable object file containing machine instructions, which is the shell program. Therefore, if I want to switch from CSH to bash, I could actually run the bash program. The effect of running the bash program is that the bash shell is going to execute, it might have a different prompt. For example, the prompt might be dollar sign. Subsequently, when I type a command, the command will be interpreted not by CSH, but by bash, because I am now executing the bash shell.

Now, if I look at the output from p s, what would I see? We saw that in the previous instance when I type p s, I got information about all the processes which my program is running. So, I expect that they would be one line for p s itself as we saw last time. They will be one line for bash, because thats the shell that I am currently using, but as you would imagine they will also be one line for CSH, because CSH is also still in execution. So, there will be three lines in the output, at least.

Now, at this point, if I am happy with having experimental with bash, I may want to get rid of bash and by the way that I can do that, is by typing in the command exit. Remember, this is the bash shell prompt, and exit is now I request to exit from bash. What will happen? The bash program will exit and I will be left with the C shell command prompt. So, just like this exit, they could a lot of other shell commands and therefore, when you type in the command, you could type in the name of any executable file, such as a dot out or bash or you could type in the name of any shell command, and one can learn more about the different shell commands by reading the manual of the particular shell that one is interested in.

Now, there are also other complications, which as you use shells, you will be aware of, but all of that functionality must go into understanding what the command is asking for, and that is a non trivial operation. But, for very simple shell, it could just be reading in the command and calling the appropriate executable file. Now, the next issue which is of some importance is, how can the shell get the command executed? For the specific example that we have, somebody the user wanted a dot out file executed. So, how can the shell cause a dot out to be executed? Now, we have seen on the one mechanism that is available, to change the memory image of a program and that is the exec system call. (Refer Slide Time: 24:30)

So, one possibility for the shell is that it could run, in order, to achieve a dot out, the shell could actually execute of an exec, indicating that it wants its new memory image to be that associated with a dot out, and that would cause the memory image of the shell, let us say the bash image to change, as we had seen in the previous class to the memory image of a dot out. So, this is now the tech segment of a dot out, and as you imagine this is not what we want to do, because the effect of the shell exactly in a dot out will be that the shell itself ceases to exit.

We no longer have a process which has its text, the instructions of that shell that has been over return by the instructions of a dot out. Therefore, in order to get the command executed, the shell cannot just call exec, the exec system call with the name of the file that is to be executed. Rather, it will have to certainly create a process using fork, as we have seen in the previous lecture, and then within the child process do the exec of the required executable file. This is a sequence which we had seen in the previous lecture. Therefore, that is in effect what the shell would have to do in order to get the command executed, if the command is the name of in executable file. If the command is just the shell command then the shell will contain instructions, will have been written to contain instructions, which will achieve the required effect such as exit.

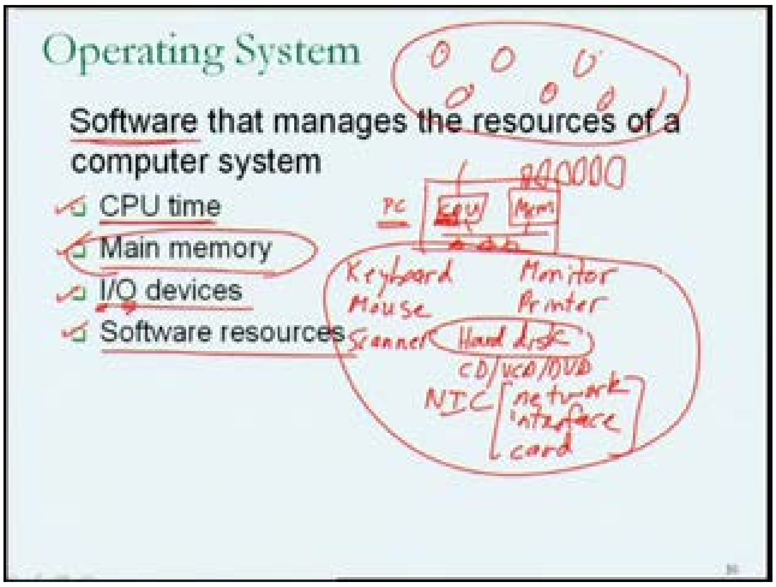
So, the bottom line is that the various steps within the shell and this is the program that you and I could write, will be making many system calls, for example, to prompt the user to type in a command, one could think of using print f. If I actually analyze what print f is doing, print f, is just the name of a function or something like that, but ultimately it calls a system call, which is capable of outputting on to this screen. The screen is I/O device, which is shared among possibly many processes and therefore, access to that I/O device must be shared through operating system functionality and ultimately a system call must be used in order to cause the output to take places to that I/O device.

Similarly, in order to read in the command, the shell will probably use something like scan f, but that is just the name of a function or something like that in a library, ultimately that function is going to call or make some kind of a system call, which will have the capability of transferring data from the keyboard, which is a hardware device into a variable in memory.

So, even for the re-command a system call will be involved, but the person writing the shell does not have to know anything about the system call involved in the outputting to the monitor or the system call involved in reading from the keyboard. The person writing the program may just use print f and scan f. In understanding what the command is asking for its possible that no system calls are required, but as we just seen in order to get the command executed, both fork and exec, may be required.

Again, if the command is shell command then they may not be the need for fork or exec, but if it is the name of the executable file, then conceivably both fork and exec will be required. The bottom line is we can see that in order to implement this functionality, one can just write a C program, making the appropriate system calls as in when needed, and this program is just in ordinary program. It is an ordinary program which will request the operating system for the key functionality as and when required using the system call mechanism, which we at this point we have somewhat understood, which we understand (( )) system call are as well.

(Refer Slide Time: 28:23)



Operating System

Software that manages the resources of a computer system

- CPU time
- Main memory
- I/O devices
- Software resources

So, at this point we will now formally say looking at the operating system in more detail. I have mentioned that the operating system is software that manages the resources of a computer system. Clearly understand that the operating system is entirely software; it is not hardware. The requirements of the operating system may have affected the way in which the hardware is designed as we have seen. It is conceivable that in designing instruction set architecture, the architect may have add to include certain instructions in order to properly support Unix, but all the same Unix itself it is just the software.

So, it is the software that manages the resources of a computer system. What do I mean by the resources of a computer system? I am going to list four types of resources. First, there is the CPU time and just a few thoughts on this before we proceed. Now, we have a situation where we have a computer in which there is one CPU, one main memory, a bus and many I/O devices. The key here is that there is only one CPU and on this they could be many processes as executing. However, there is only one CPU and if there is only one CPU, there is only one program counter. In other words, only one instruction can actually be executed, at the given point in time and the current understanding of how the hardware works.

So, what then does it mean to have many programs in execution at the same time or concurrently? Very clearly the operating system is managing the CPU time. It is allowing, let us say, one program to execute now and after sometime allowing another program to execute. We will look into some details of that little bit later, but we have to view the CPU or the time of usage of the CPU as being one of the hardware resources that has to be managed by the operating system. There is nothing in the hardware to manage the CPU across processes.

The hardware does not even know about the existing the processes. The hardware just knows about instructions and the execution of instructions. So, the management of CPU time is going to be an important functionality of the operating system. Secondly, I have listed main memory. What we know about main memory? We know that as for as main memory is concerned, if there are many process, as in execution, then all of those processes must have their programs resident in the main memory or present in the main memory.

So, as for as main memory is concerned, I will have the main memory image of each of those programs present. That is the only way that each of those programs could potentially be executed within the CPU, because its instructions have to be fetched from main memory, not from elsewhere. Therefore, very clearly the operating system must be involved in the management of main memory.

We have seen that there is one component in the CPU called the MMU, which assists in something called address translation. But address translation, alone might not be the whole story as far as management of the main memory is concerned. Address translation might be the key lowest level link in the chain, which must be implemented in the hardware, which is why it is represented by the box called MMU. The rest of memory management, the rest of the management of main memory is something that as be of importance.

Next, I have listed I/O devices. Now, this is slightly more complicated story, because I have not talked in detail about I/O devices yet. This might be the right time to do that. You will remember that I/O devices include devices for input and devices for output, and we have used many examples. For example, we understand that one of the important mechanisms for input is to keyboard, there is also the mouse on the output side, there is the monitor, but there are other kinds of devices. For example, we are aware that I have talked about the fact that they could be disk memories, hard disks inside a computer system. Is that input or an output device? So, I will put somewhere in between the notion of hard disk, you may have C D drives, V C D drives, whatever, in a computer system. These are other examples of I/O devices, they may be a printer; a printer is typically I/O output device. They could be scanners. These are all examples of I/O devices, but there are others as well.

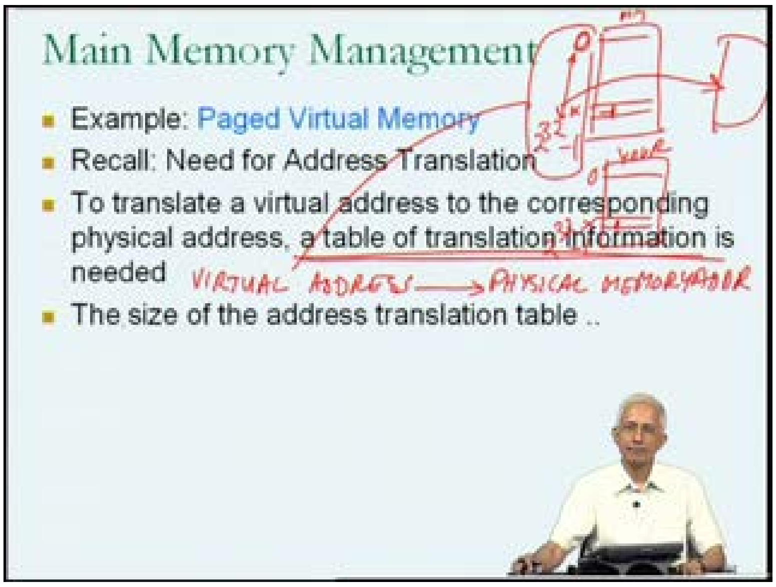
For example, what about the network interface card that is present in your computer. I will refer to as an N I C, many of the computers that you have today are connected on networks and the connectivity is provided through a piece of hardware, which might be call the Network Interface Card, could be Ethernet card or could be something else or could be provided through wireless interface. But, this is just an example of a network interface card and one may have to view the network interface card also as being an I/O device, since input and output from this particular computer does take place through the network interface card.

So, when we talked about I/O devices, the range of hardware that could be included could be fairly vast; different kinds of functionality, and we may not be able to address issues, related to all of these, and in may not in fact, be important to us. We would want to concentrate on those I/O devices which play a clear, specific and particular part, in the day to day execution of kinds of programs that we might execute, and that might in fact, just be restricted to the hard disk.

So, among all the I/O devices, we may just restrict our attention to one or a small number of these in our discussion. Now, last but not the least, the operating system will possibly have to be involved in the management of software resources. We have already seen in our diagram of the computer organization, the software side that they are many processes in a computer system. Many programs are in execution, represented by the software entities or the software resources called processes, which requires the operating system to intercede on its behalf. You have also seen that the operating system is involved in the creation, or the termination or the manipulation of these processes.

Therefore, from the operating system perspective, the processes and other objects of that type or software resources that have to be managed. Therefore, in studying about the operating system or in discussing the operating system, we need to look at each of these four (Refer Slide Time: 34:55), one after the other, to understand how the programs that we write and execute or effected by the operating system, in terms of, the way the operating system manages CPU time, main memory, I/O devices or software resources. So, we will need to go through this one by one, and I am going to start with main memory.

(Refer Slide Time: 35:15)



Main Memory Management

- Example: Paged Virtual Memory
- Recall: Need for Address Translation
- To translate a virtual address to the corresponding physical address, a table of translation information is needed
- The size of the address translation table ..

So, this sub topic is what we normally will call is Main Memory Management. It is the story about how the operating system is involved in the management of main memory For future reference, the specific example, which I am going to use with something, which is fairly popular among kinds of operating system that we are dealing with is something known as, Paged Virtual Memory.

Now, let me just remind you that we have seen that there is a need for address translation. Where did this need arise from? We saw that on any particular computer system, one particular computer system, there could be many programs in execution. Each of these programs is written or compiled to assume that it chooses addresses from 2 to the power of 32 or from 0 up to 2 to the power of 32 minus 1, and that they could many such programs in execution at the same time, and therefore if all of these programs had their memory images in memory at the location specified, then one program would certainly be able to effect another program.

Not only, by the fact that their instructions would be overlapping, but that if I knew that if this is the memory image of my program, and this is the memory image of your program, and I know that your program has in important variable at a particular memory address, then I could modify; your programs thinks the value of the variable is one, I could values of I could change the values of the variable to minus one by just changing the variable from my program.

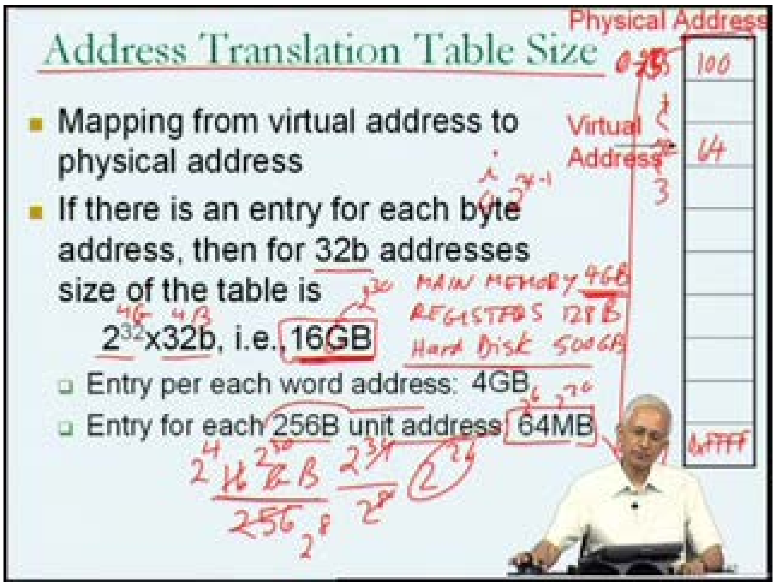
So, there is a need to protect one program from another program, in terms of its memory image, and that is why we figure out that address translation was a mechanism to do that. The idea of address translation was that the addresses 0 to 2 power 32 minus one that the compiler assumes, will be associated with the program in execution, will be viewed as being specific to that process or that program in execution, and the prior to actually being sent to main memory, during program execution, the address would be translated to an actual memory address, and this translation would be one by a piece of hardware call the MMU.

Now, we did not talk about how this translation was going to be done. We just said that it would be done by the MMU. Now, if you think about little bit, in order to translate, one address into another address, one will have to have the base information on which the translation is to be done. In other words, one will need a table of translation information. If there is a table, which tells the MMU what address 0 relating to process 0 means, as far as main memory is concerned, then the MMU can just change address 0 into that address, by doing the look up in the translation table.

Therefore, it is the useful idea to have; the ideas that address translation can be implemented using a table of translation information. What will this table contain? This table will contain, for each address in the range 0 to 2 to the power 32 to 0 to 2 to the power 32 minus one, will contain the corresponding address, in main memory, to which a translation is to be done.

Now, this of course, raises an issue and that is the issue of the size of the address translation table. At this point, I should possibly remind you that we had talked about virtual or artificial addresses, and the addresses generate by the compiler are virtual addresses as suppose to physical or real main memory addresses, and that the address translation process done by the MMU basically converge so virtual address into a physical memory address or physical address. Now, if this translation is going to be done, using an address translation table, we have to think a little bit about how big that address translation table is going to be. Let us spend a little bit of time thinking about that..

(Refer Slide Time: 39:25)



Address Translation Table Size

- Mapping from virtual address to physical address
- If there is an entry for each byte address, then for 32b addresses size of the table is
  2^{32}x32b, i.e. 16GB
  - Entry per each word address: 4GB
  - Entry for each 256B unit address: 64MB

So, current concern is the size of the address translation table. Now, as I reminded you the address translation table contains a mapping from virtual addresses to physical addresses. It tells the MMU how to translate each virtual address into the corresponding physical address, and one way that this could be done is, as I said through a table. This table is going to contain for each virtual address the corresponding physical address.

So, in the diagram that I show you here I have actually showing you a table, which contains a lot of physical addresses. How many physical addresses? That depends on how many virtual addresses there are. So, let us suppose there are 2 to the power 32 virtual addresses, and then the first entry in this table will tell you how to translate address, virtual address 0, into a physical address. Similarly, the last entry in this table will tell me how to translate the address 2 power 32 minus one into a physical address. How will it tell me that? It will tell me this by containing the corresponding address. For example, suppose virtual 0 is suppose to translate into physical address 100, then the first entering in the address translation table will contain physical address 100.

Similarly, if address 2 power 32 minus one, is suppose to translate into virtual address 2 power 32 minus one, is suppose to translate into physical address hex F F F F, then that is what that entry in the address translation table would contain. In other words, what I am doing is the way that I use this table is, if I want to translate an address i and i is some value between 0 and 2 power 32 minus one, then all that I have to do is I have to index into this table, using address i and that will tell me how the virtual address i is suppose to be translated.

For example, if it is suppose to be translated into 64, then this entry in the table will contain the address 64. Therefore, we can calculate what the size of this address translation table is going to be, using this idea. The idea essentially is that I have a table, which tells me the physical address for each possible virtual address. The table need not contain both the virtual address and the physical address. It need contain only the physical addresses and I can by looking at the ith entry in the table, I will know the translation for virtual address i.

So, how big will this table be? Let us suppose, for the moment, that we do what I have just described. I have an entry in the table for each address that I have in terms of the virtual address page. In other words, I have one entry in this table for each byte address. So, if I have one entry in this table for each byte address, then the size of this table would be the number of byte addresses, which is 2 to the power of 32 multiplied, by the size of a physical address, and up to now, we have been assuming that the size of addresses is 32 bits. We will continue to make that assumption. What is the size of the table? The size of table will be the number of entries which is 2 power 32 multiplied by the size of each entry, which is 32 bits.

So, this is a size of the table. How big is this? We know that 2 to the power 30, is what we call the gigabyte. I am sorry Its capital G, therefore, 2 to the power 32, is going to be 4 G, 32 bits is 4 bytes and therefore, the value that we see, in other words, 2 power 32 multiplied by 32 bits, is equivalent of 16 gigabytes 4 G multiplied by 4 B, which is 16 GB, and once again remember that G, in this context, just stands for the constant 2 power 30, which is for, I could write 4G and multiplied by 4B, and come up 16 GB, meaningfully.

So, the size of this translation table is going to be 16 gigabytes. Now, this is bit of a concern, because in passing when I talked about the different kinds of memory, which are available in a computer system, I talked about main memory. We talked about registers, talked about cache, but not in enough detail to include in this list here. Then we talked about other possibilities like hard disk, DVD, CDs and so on. I also gave you some idea about how big each of these would be.

I have indicated that for example, just think about the MIPS one registers, we know that the MIPS one contains 32 bit registers, which means that the size of the registers in the MIPS one is something like 32 multiplied by 4 bytes. So, 128 bytes, which is nowhere, close to this. I had also mentioned in passing that the typical size of main memory for our very healthy machine today could be something like 4 gigabytes, the desktop or laptop that you have may have 4 or less gigabytes. In other words, this translation table is so big that it would not even fit into main memory.

We know that the size of hard disks significantly bigger, for example, many of you will have hard disks which are 300, 500, 250 gigabytes in size. Therefore, this translation table is so big that the only place we could store the translation table, on the computer system, would be on a hard disk, which is not very promising beginning, because we want to use the translation table as the reference for the MMU, in other words, a part of the hardware, a part of the CPU, to do translations of addresses before the addresses went to main memory.

And if this as If reference to hard disk has to take place for every address translation, we understand that the hard disk is significantly slower than main memory then this it is not going to be a very good mechanism. Therefore, it is not feasible to do things along these lines. That is the bottom line. Our concern about the address translation table is real and that if you were to use in address translation table, in which there is one piece of address information for every byte, then the table would be so big that would not be a feasible mechanism for address translation.

Now, what if on the other hand, I would to assume that there is an entry in the table for each word address, so rather than for each byte address, I have a translation entry in my address translation table for each word address. How is this going to change things? Now, rather than having a separate address translation table for byte 0 entry, for byte 0 byte 1, byte 2 and byte 3, there will be a single address translation table entry for bytes 0 through 3. In other words, the size of the address translation table is going to come down by a factor of four.

So, the size of the address translation table will be 4 gigabytes. This is little bit better than it was before, because now the address translation table could fit into a large main memory, but it would occupy all of main memory, and therefore, once again it does not seem like a feasible option, because remember our starting point was, we want the problem, the need for address translation arises, because there are many programs resident in main memory at the same time, and if all of main memory is full of the address translation information for one of those programs then system does not work.

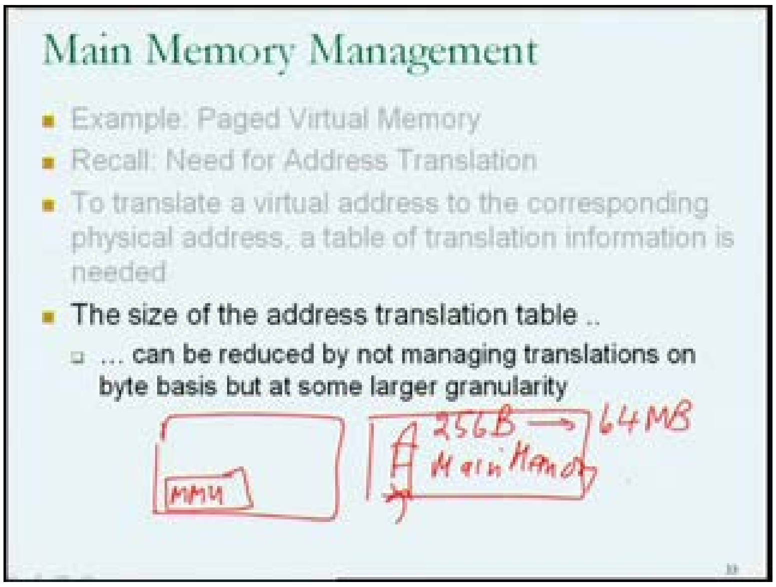
So, clearly we need something else. Now, let us look at the a larger grain possibility. Let us just suppose that we started off assuming there was one address translation entry for each byte, and saw that was not feasible. We then said, let us may be at one address translation information for each four bytes and saw that too was not workable. Moving in the same direction, let us assume that there is one address piece of address translation for each 256 bytes. In other words, for all the bytes between byte 0 and byte 255, I have just one entry in the address translation table.

So, the size of this is going to be one by 256th of the size of the original table that we had and what is 16 g b divided by 256? The answer is just remember that g is 2 power 30 and 256 is 2 power 2,4,8, 16,30, 264, 266 is 2 power 8, I believe. Then what you end up with is 16 is 2 power 4, from the top we have 2 power 34, on the bottom we have 2 power 8.

We left 2 the power of 26, which is 64 M. M is 2 power 20 and 64 is 2 to the power of 6. So, that works out correct. So, the reason, I went through this example is so that you will get use to the notion of dividing one value which is in this notation by another value in the same notation, typically the powers of two cancel out so the calculations are easy.

So, under the assumption that there is one entry in the address translation table for each 256 byte units of the virtual address space, then the size of my address translation table is only 64 mega bytes and therefore, I could comfortably include many such address translation tables into main memory, for the many programs that are in execution, at a particular point in time. So, we seem to be moving in the right direction, but the key towards moving in that direction is that we need to associate with the address translation table entry, a very high granularity of address, not a single byte address, not a single word address, but, a single virtual address for at least 256 bytes, consecutive bytes in the virtual address space

(Refer Slide Time: 48:54)



Main Memory Management

- Example: Paged Virtual Memory
- Recall: Need for Address Translation
- To translate a virtual address to the corresponding physical address, a table of translation information is needed
- The size of the address translation table ..
  - ... can be reduced by not managing translations on byte basis but at some larger granularity

MMU
A
256B → 64MB
Main Mem

So, the size of the address translation table was a concern to us and we saw that the means of reducing the size of the address translation table was by not managing translations on a byte basis. In other words, not a separate entering the address translation table for each byte address, but some larger granularity, such as one address translation table entry for each 256 bytes of the virtual address space.

So, this seems to address the problem of the size of the address translation table. Now, this is not fully address a problem that we have, because you understand that all of this translation supposed to be done by the MMU, and by using this idea, we have reduce the size of the address translation table to 64 mega bytes, which I argued was feasible, in the sense that, I could have many such tables present in main memory.

But, if the address translation table is present in main memory, then in order to do the translation, the MMU will have to make a main memory reference, which is going to take towards the magnitude more time than we hoped it would take, because we were including the MMU inside the processor, hoping that the translation could happen a processor time lines speeds, not at memory time line speeds.

So, this is still a little bit away from being a feasible solution to the problem of how to manage main memory and when we continue from this point in time in the next lecture we will understand little bit more about how this scheme can be made to work. This would remind you what we have been looking at in todays lecture. In todays lecture, we have formally understood that we need to understand the operating system, from the perspective of four aspects of its functionality, how it manages CPU time, how it manages main memory, how it manages I/O devices of which we may look at the specific example of the hard disk, and how it manages software resources on the computer system such as processes.

We also looked at the mechanism that is provided for safe transfer of control from a process into the operating system through the functionality of the system call by providing inside the instruction set, and inside the hardware, the capability for execution in multiple modes, one of which is a privilege mode, in which only privileged instructions can be executed, as well as a user mode ordinary instruction, through which in ordinary user process can be switched to the privilege mode, in order to call a System call, and this instruction in the MIPS one instruction set was our sys call instruction. So, with this we are all set to formally look how different functionalities of the operating system affect the programs that we execute.

Thank you.