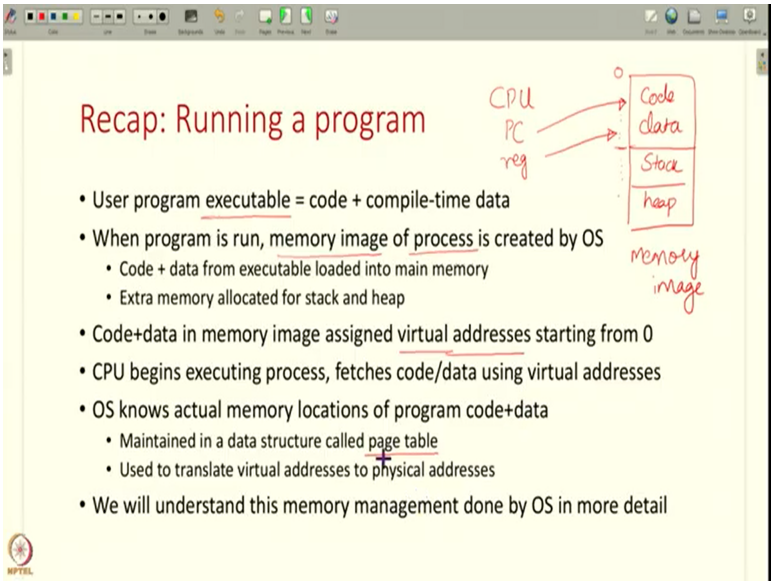
Design and Engineering of Computer Systems
Professor. Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 11
Memory Management in OS

Hello everyone, welcome to the eleventh lecture in the course Design and Engineering of Computer Systems. In the last week, we have studied process management in operating systems, we have seen how the operating system runs user programs on the underlying hardware. And we have also seen how programs can be run inside virtual machines or containers. Starting this lecture onwards in the next few lectures, we are going to understand memory management and operating systems.

(Refer Slide Time: 00:43)



So, this slide has a recap of what we have already studied in the previous lecture where we introduced operating systems. So, we have seen that whenever you as a user write a program this is compiled into an executable, this executable has the code as well as some data that can be allocated at compile time.

And whenever this program has to be run, the operating system creates a process and creates a memory image of this process in RAM. So, in the main memory, in RAM, you will have the memory image of the process. So, this memory image of the process has code data and this is there in the executable the compile time data and the operating system also creates a stack, a

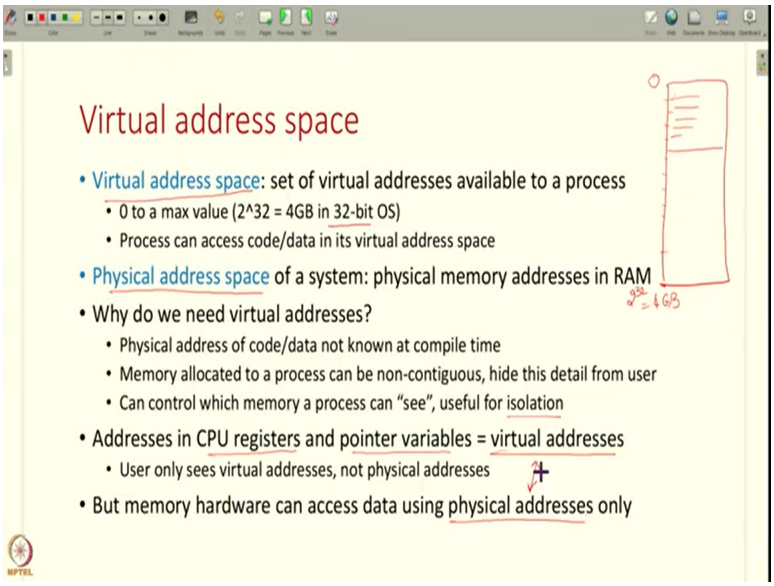
heap and other components in the memory image. And all of this memory is allocated in RAM so that the process can run on the CPU.

Now, one thing to note that we have already discussed before is that all of these instructions and data are assigned addresses. Now, what addresses do you assign? The compiler assigns addresses starting from 0 to the code and compile time data, then whatever is the last address from there for the other components like stack and heap also, the OS assigns addresses.

So every instruction, every variable in your memory image has an address and these are called virtual addresses. Why because these do not actually correspond to real memory addresses in RAM, they are just addresses we assigned for convenience. But in reality, this memory image might be placed at some other location in RAM and that location the operating system knows these actual locations, why because the operating system is allocating memory for this program. And therefore, it knows those actual memory addresses of these instructions and data.

And this information is maintained in a data structure called the page table. So whenever the CPU is executing an instruction, say the program counter has some address of some instruction, and it tries to fetch this instruction, various registers try to access the data in a program. All of these accesses, the CPU accesses the memory using virtual addresses, but they have to be actually translated into physical addresses for the memory hardware to actually return the data. So, how is all of this done? What are virtual addresses? What are physical addresses? How are they translated all of this we are going to understand in more detail in this lecture.

(Refer Slide Time: 03:22)



So, let us first understand the term virtual address space. So, the virtual address space of a process is the set of all virtual addresses available to a process. So, every process can allocate virtual addresses starting from 0 to some maximum value to whatever instructions and data it wants to use, starting from 0 to this maximum value is basically dictated by your CPU.

If your program counter in your CPU is 32 bits, if your architecture is 32 bits, then you can store a maximum address of 2 to the 32 which is 4 gigabytes, this is the maximum address you can store in your program counter. Therefore, this is the maximum address space that you can have. So, this is the virtual address space of a process which contains all the code and data that the process can access.

But, there is also what is called the physical address space of a machine. These are the actual physical memory addresses that the RAM has. And these two are not the same thing, for example, if a process has a virtual address, some code at virtual address 0, it does not mean that this is that RAM address 0, in RAM, it could be at some other physical address.

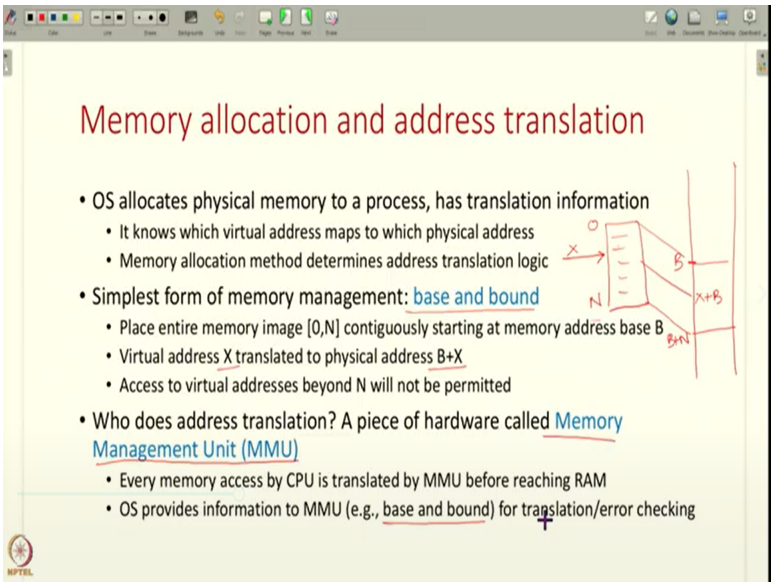
So then, you might ask why do we need these virtual addresses? Why are we using these two different addresses? Is not it confusing? The reason we need virtual addresses is that you cannot assign physical addresses to code and data at compile time because it is not known. When the compiler is generating the code and the data in the executable, it stops assigning addresses from

0 because it does not know where in RAM will this code and data be placed where will this process reside in RAM that is not known.

So, therefore, it is convenient to use virtual addresses. And there are also other reasons, for example, all the memory of a process may not be allocated contiguously in RAM a small part might be here a small part might be there you do not want the user to know all those details about the physical addresses. And this also, this virtual address space also gives you some kind of isolation you can control what are the addresses what are the code and data that a process sees? we will understand in more detail how isolation is done, we will see that later in this lecture.

But anyway, the summary is that everything that the program sees whatever addresses you store in pointer variables, whenever the CPU requests a piece of data or an instruction from an address, whatever addresses are stored in CPU registers, all of these are always virtual addresses and all of these virtual addresses that a process sees and stores code and data at that is called the virtual address space. But in reality, the memory hardware the RAM can only access data using physical addresses. So, this translation is needed.

(Refer Slide Time: 06:22)



So, now, the question comes up, how do we translate these virtual addresses into physical addresses and who can do this translation? So, the operating system knows where a processor is located in RAM it knows the physical addresses, so, the operating system has all the translation information. And this translation information that the OS has can be used for address translation.

So, let us understand how this translation happens using a very simple example. And of course, how you do the translation actually also depends on how you allocate memory, there are different ways of allocating memory that will result in different address translation logic.

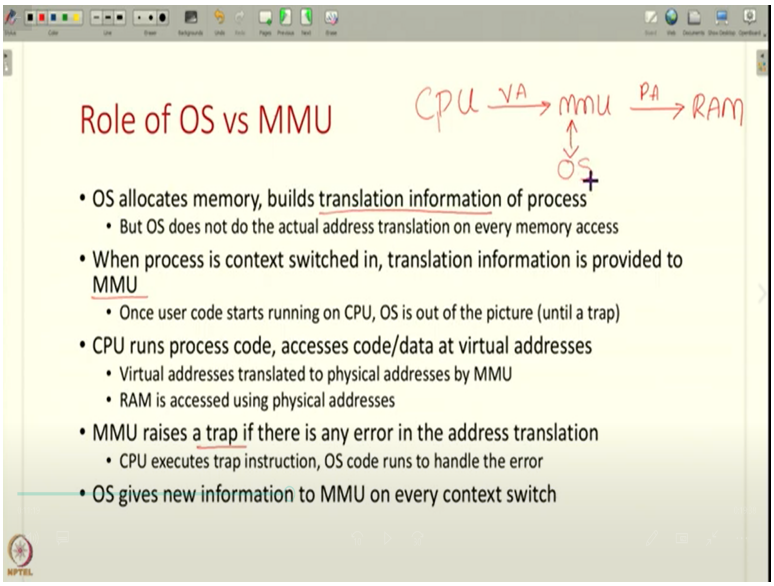
So, the simplest way of memory allocation is what is called the base and bound that is, if a process has say virtual addresses from 0 to n, it has a memory image of size n, then what the OS will do is in RAM at starting at some location B at some base value B to B plus N, it will simply store the memory image of the process. This is called a base and bound method; this is the base and this is the bound that is, you will place the entire memory image contiguously starting at some memory address B.

If this is your memory allocation logic, then how will you translate a virtual address to a physical address if the CPU requests anything at some virtual address x, then this address x simply maps to address x plus B, you simply add the base and you will map to address x plus B in RAM virtual address x is translated into physical address x plus b it is simple this address translation logic is simple.

And one more thing we have to check this if the process requests some address beyond n, then we should not translate it to a physical address you simply cannot keep adding B everywhere you have to check the bound also because a process can only access its memory image. So, the upper bound also will have to be checked. So, this is the address translation logic. So, who does this address translation logic? We have a special piece of hardware on the computer called the memory management unit that does this translation on every memory access.

That is, whenever the CPU requests instruction or data at address x, the MMU will translate it into address B plus x and then the RAM can fetch the data at address B plus x. And this information of what is the base what is the bound that is provided to the MMU by the operating system, but the actual translation is done by the MMU. In this way, the functionality is split between the OS and the MMU.

(Refer Slide Time: 09:16)



So, to summarize, the operating system allocates memory and builds the translation information whatever information is needed the base bound that is provided by the operating system, but the operating system does not do the actual address translation. Why? Because the operating system is not running all the time. We have seen in a previous lecture that once the CPU starts running a user process just the user code is running on the CPU.

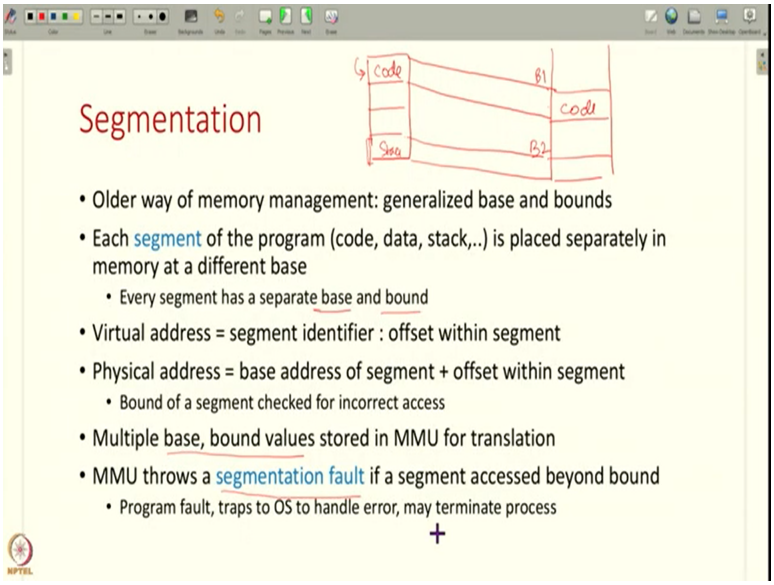
The OS is not in the picture only when a trap happens then the OS comes into the picture. So, therefore, the OS itself does not do the translation cannot do the translation on every memory access. Instead, this translation is done by the MMU using the information using the translation information provided by the operating system.

So, whenever a processes context which then this translation information corresponding to that process is provided by the OS to the MMU, and then the OS of the picture until a trap occurs, and whenever the CPU is running the process, fetching instructions fetching data all of that is happening with the address translation being done by the MMU. And these virtual addresses are translated by the MMU and the RAM is then accessed using the physical addresses.

So, the CPU gives a virtual address, the MMU translates it into a physical address and then the actual RAM is accessed using the physical address. And if the MMU runs into any issues, for example, the process has accessed an address beyond its bound in such cases, the MMU will raise a trap and you will go back to the operating system.

So, the role of the operating system is to provide this translation information to the MMU and update this information on every context which whenever it needs to be updated, this information is updated. So, the OS will talk to the MMU and provide this information and the MMU will come back to the OS in case of any problem, but otherwise, the translation is handled by the MMU.

(Refer Slide Time: 11:27)



So, another memory management technique so, we have seen a simple base and bound; a slightly more complicated system is what is called segmentation. You can think of this as a generalized version of base and bound that is instead of placing the entire memory image as one contiguous entity in RAM, what we will do is we will split this memory image into code, data, heaps, stacked into different segments and each segment will be placed in a different location in RAM.

For example, this code segment could be placed here and you know say the stack segment could be placed somewhere else. So, each segment is placed separately at a different base. So, this could be the base of the code segment, this could be the base of the stack segment and so on, every segment is placed separately in RAM, and each will have a different base and a different bound.

Why do we do this, it gives us more flexibility and instead of placing this big chunk of memory image in memory, we will place it in smaller chunks might make it easier to find free locations and so on. So, every segment has its own separate base and bound, then a virtual address is

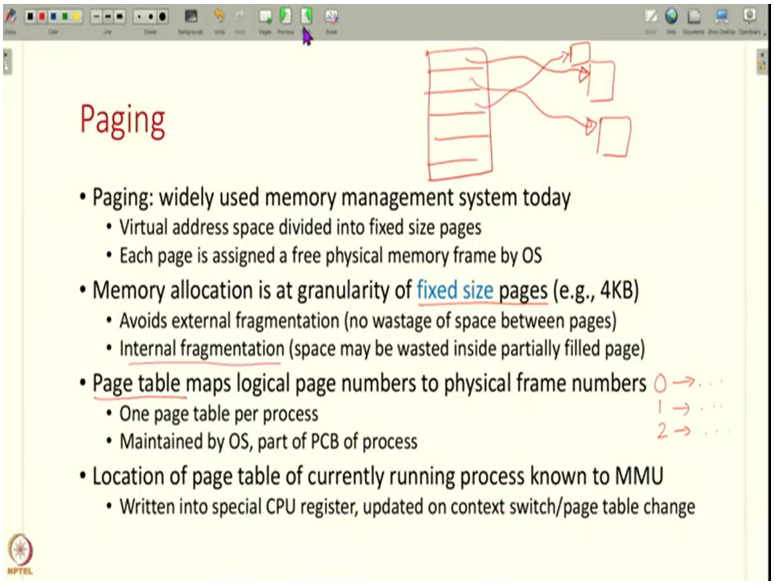
basically which segment you are in and what offset into the segment you are. So, you have how do you translate this to a physical address you will add the base address of that segment to the offset, the base address plus the offset will give you the physical address.

So, if you are using segmentation, if the OS is allocating memory to a process using the segmentation logic, it will give multiple of these base and bound values to the MMU for translation. And the MMU will see when the CPU requests a certain virtual address it will see which segment what is the offset at the corresponding base and return the physical address.

Now, the MMU also if you access beyond the bounds of a segment, if you access some data beyond whatever is your limit, then it will throw a fault which is why those errors are called segmentation faults. Even though segmentation is not widely used today, there still is called a segmentation fault.

So if you access an array out of bounds, you will get a segmentation fault why because, you have gone beyond the limit of your segment. And when such faults occur, it will trap to the OS and the OS will handle the error. And it may also terminate the process. If you are accessing memory that you should not be accessing you can be terminated.

(Refer Slide Time: 14:09)



So next, the way of managing memory is what is called Paging. So all of these things like segmentation and base and bounds are all simple ideas, but modern systems predominantly use paging. That is what is paging, you will divide your virtual address space into fixed size pages.

So, you are no longer dividing at the boundary of code, data, stack, heap, for example, your code could be only half a page, your stack could be two pages does not matter. You are dividing it into fixed size chunks called pages. And each of these pages is allocated a physical frame in memory, a fixed size physical frame in memory.

So, virtual address space consists of pages and each page is assigned a free physical memory frame in RAM. So, what is the advantage of doing memory allocation at this fixed granularity of pages? So, typical page sizes say 4 Kb what is the advantage, the advantage is that you do not have external fragmentation that is your memory just has you know fixed size chunks like this there is a page here a page here a page here, you do not have any gaps between pages.

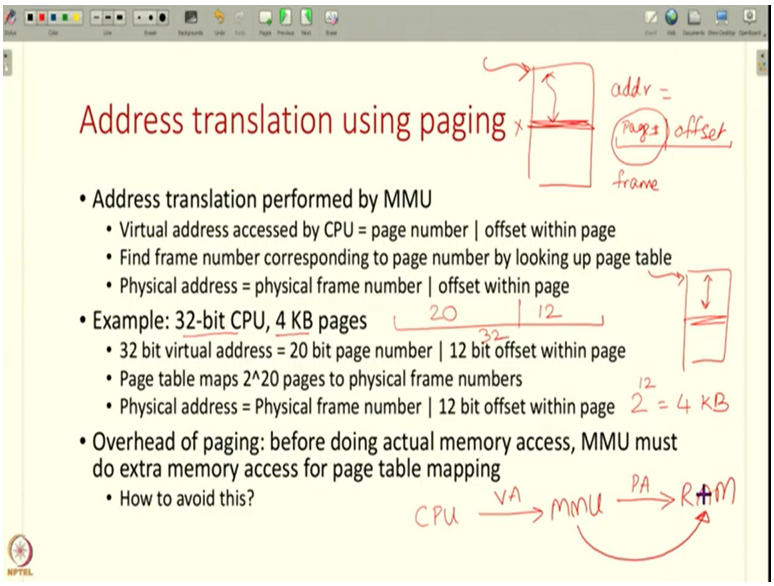
If you did variable sized allocation, then what would happen, you have a chunk allocated like this then another chunk allocated like this then you might have a gap here another chunk you will have these small holes left in between it will become very messy. So, if you do not want to do that, then you will do a fixed size memory allocation.

But, the problem with this fixed size memory allocation is that there could be internal fragmentation that is inside a page some space could be wasted. For example, if your pages 4 Kb or processes only using 2 Kb the other 2 Kb is free that is the price you have to pay for doing fixed size allocation. Now with paging, the OS maintains a data structure called a page table.

So, if you look at the virtual address space of a process it is divided into pages and if page 0 is placed at some physical frame number say 42 in memory, page 1 is placed that another physical frame, page 2 is placed at another physical frame. So, all of these numbers page 0 is placed that some frame, page 1 is placed at some frame, page 2 is placed at some physical frame all of this information is stored in the page table of a process.

There is one such page table for every process and this is part of the PCB of the process and it is maintained by the operating system for every process. And this page table is provided to the MMU whenever a processes context is switched in so that the MMU can use this page table in order to do address translation. So, this location of the page table is known to the MMU and MMU will use it, it is in fact written into a special CPU register and this is used by the MMU for address translation.

(Refer Slide Time: 17:24)



So, how does address translation happen with paging? If you look at you know, any page and inside a page there is some data at certain offset within the page. So, then your address this address x of this piece of data will actually you can split it into two parts, you can split it into a page number and the offset within the page. So, you will get some page number and then you will get some offset within the page. So, this is your virtual address.

So, how do you translate it into a physical address? You will take this page number, you will look up the page table find out which frame which memory location is this located at and then you will add the offset. Once you get the frame number, you know the frame number, then you add the offset and you have gotten your physical address that is how address translation is done by the MMU.

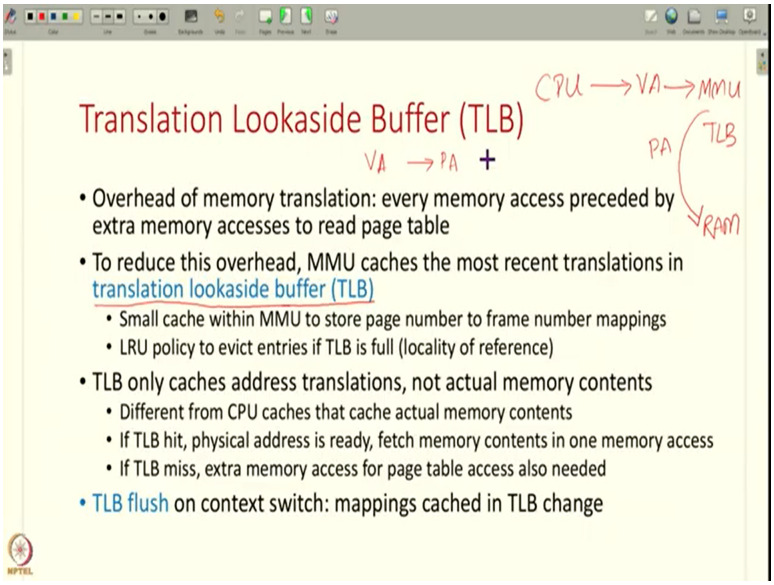
For example, if you have 4 Kb pages on a 32 bit CPU in your 32 bit address for a 4 Kb page, how many bits are needed to specify the offset, you will need logarithm of 4 Kb, those many bits you will need to specify the offset. Therefore, in this 32 bits, the last 12 bits will be the offset within a page why is that because 2 to the 12 is equal to 4 Kb.

Therefore, with 12 bits you can specify any offset within a 4 Kb page and the remaining 20 bits will be your page number. So, whenever the MMU gets a 32 bit virtual address, it will split it into two parts. Take this 20 bit page number, look it up in the page table, find the physical frame

number and then add the physical frame number and the offset concatenate them to get the physical address. This is how address translation is done by the MMU.

But, there is an overhead involved here before doing every memory access, so the CPU wants to do a memory access it has provided a virtual address to the MMU. And before the MMU can actually provide a physical address and access RAM, it must first go to memory, get the page table read it do this translation and only then we can access memory, before doing a memory access. You have an extra memory access to do to access the page table, that is an extra overhead you are somehow doubling the amount of memory accesses you have to do which is not ideal. So, how does the MMU avoid this extra memory access?

(Refer Slide Time: 20:05)



What it will do is it uses a cache again, this is a principle that we have seen, often, it uses a cache of these address translations. Every time a virtual address is translated to a physical address. What the MMU will do is it will remember this translation, it will remember the page number to frame number mapping in a cache is known as the TLB or translation lookaside buffer.

So, this is like any other cache, you will use the least recently used policy to evict entries if the cache is full. Of course, you cannot store these translations for every possible page, you can only store the most recently used translations, and the older ones you will keep evicting them just like any other cache. But this TLB basically helps you do this address translation faster.

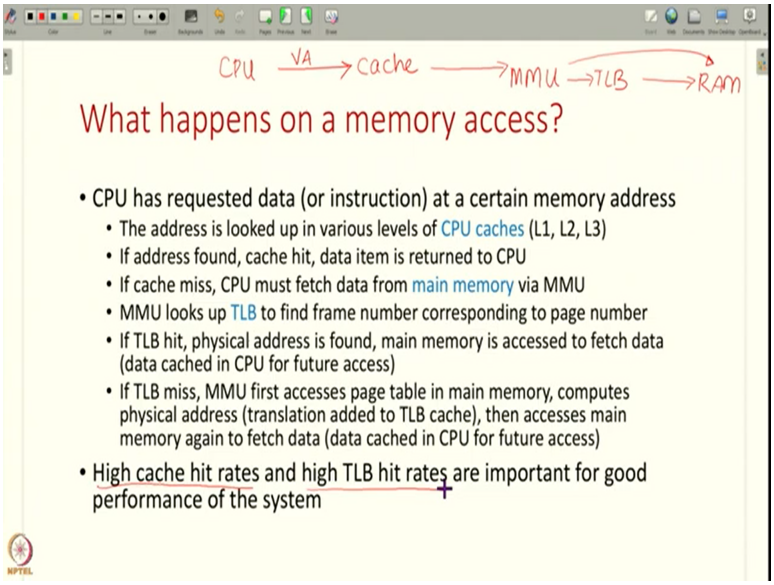
Every time the MMU wants to translate an address, it will first check the TLB. If the translation is there, you can directly access the memory content without first going to memory to get the page table. So, note that this TLB is not like the CPU cache. The main difference is that TLB only caches the address translation, not the actual memory content that is there at that address.

So, if your CPU wants to give a virtual address and ask, ask the MMU to translate this virtual address, then the MMU still has to go to RAM to fetch the data, just that with the TLB, you will directly know the physical address to go to RAM. You do not have to first go to ram read the page table and then go to ram again, to fetch the data, you are avoiding that extra page table access.

But, the TLB does not store the actual memory content itself, you still have to go to RAM, it only provides you with the physical address. So in this way, it is slightly different from a CPU cache. So if there is a TLB hit physical address is ready, you can just do one memory access and get the data you want. If there is a TLB miss, you have to do extra memory accesses for the page table. So which is why the TLB is important.

And the TLB entries, all of these mappings also have to be cleaned up and have to be flushed every time there is a context switch because of the new process, new mappings. So, you have to keep these up to date with whichever processes running on the CPU. So now, we have all the pieces together to understand what happens on a memory access.

(Refer Slide Time: 22:38)



The CPU has requested some data using a virtual address. The first thing that is checked is you will check the CPU caches. If this address is present in the cache, then you have your memory content you have your instruction or data whatever you want, and immediately the CPU can start executing the instruction or processing the data you do not have to do anything else if it is a cache hit.

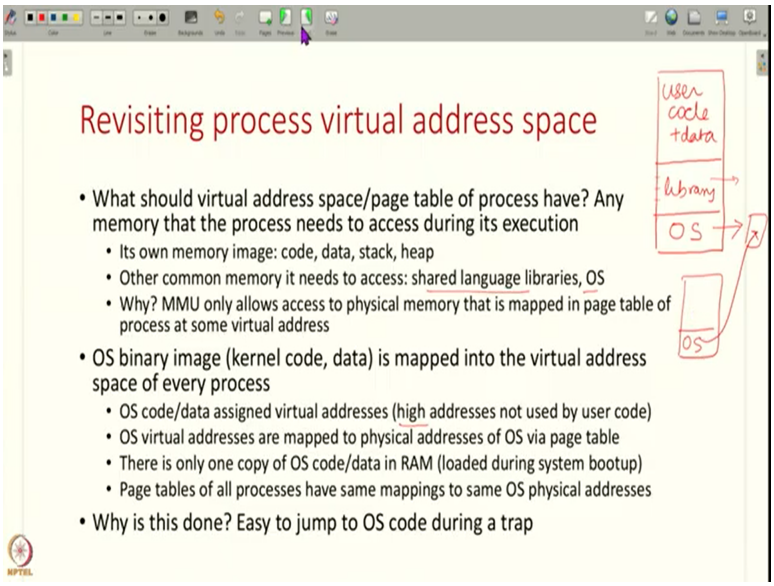
But if it is a cache miss, then you will go to the MMU and then what the MMU will then do is it will check with the TLB this virtual address to physical address mapping do you have it. If the TLB has it, if it is a TLB hit, then you will directly go to RAM and you will fetch the data at a physical address because the physical address is known to you.

If it is a TLB miss, then the MMU will first go to RAM access the page table translates the address maybe store a copy of it in the TLB for the future and then it will go to memory again to get the memory contents at that physical address. So, there are multiple memory accesses first to access the page table for address translation information and then actually address access the RAM at that particular address. And then, the contents of this memory will be returned back to the CPU, CPU can store it in the cache for future accesses.

So, in this way, there are many steps involved in memory access, and at each point, there is an extra overhead. If there is a cache miss, then it takes a long time if there is a TLB miss, it takes some extra effort to translate the address. Therefore, having high cache hit rates and high TLB hit rates, these are important for good performance in your system.

So later on when we are studying how to optimize performance. In that module of this course, we are going to revisit these things and see how we can ensure high cache hit rates and high TLB hit rates. So, the one small thing that I want to tell you before we end this lecture is let us understand this virtual address space a little bit better.

(Refer Slide Time: 24:56)



What all should a process have in its virtual address space. Of course, all the user space code and data in your program stack, heap everything you should have in addition to this, whatever other common memory that does not belong to the process, but which the process will access all of these are also assigned virtual addresses in the virtual address space of a process. Things like the shared language libraries, operating systems, all of these.

So, libraries, OS any common memory that does not belong to this process itself, but will be used by the process, these are all mapped into the virtual address space of a process that is the page table will have entries. For example, the page table will have entries mapping certain virtual addresses assigned to the OS to the actual OS code in RAM. So, some virtual addresses the high virtual addresses that the program does not use for its code or data, those high virtual addresses are assigned to operating system code and data and the page table of a process will translate these to the actual operating system code located in RAM.

Note that these mappings are present in the page table of every process. That is every process will assign virtual addresses to OS code and data but there is only one copy in RAM. So, a different process virtual address space also, the addresses assigned to the operating system will point to the same physical location of the operating system code.

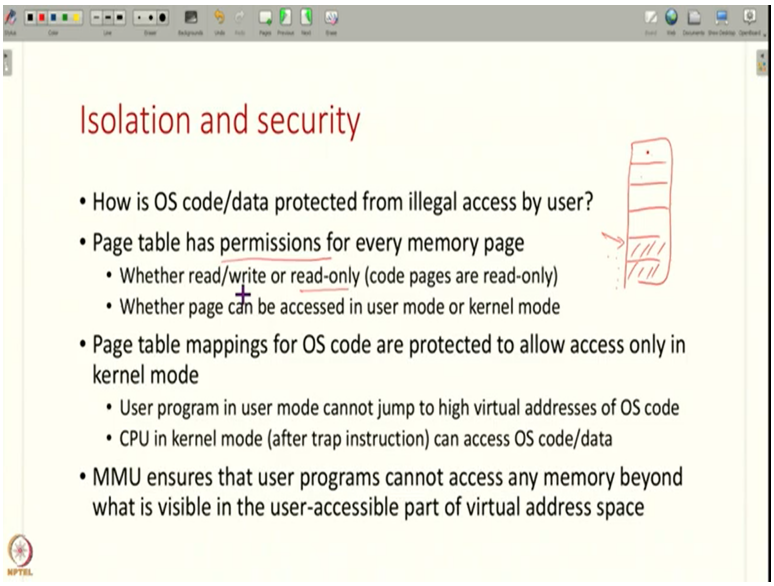
That is, the operating system is mapped into the virtual address space of every process. There are page table entries, which map certain addresses assigned to OS code and data to the actual

physical memory of the OS code and data. So why are we doing this, we are doing this because it is easy to jump to the OS code during a trap.

If along with your program Code, Data, you see the operating system, program code and data also add virtual addresses, then you can simply easily jump to them you can use the same page table and jump to OS code jumped to library code everything, of course, there will be permission checks and all of that, but it makes it easy to access all the other data that is not part of your program also.

Therefore, if you look at the virtual address space of a process, not just the code and data of that process, but a few other things that the process needs, they will also be assigned virtual addresses and the page table will map them to actual locations where these things are located in memory. And these can be common ones, it is not like every process will have a separate library these can be common physical memory, that is whose address is present in the page table of every process.

(Refer Slide Time: 27:49)

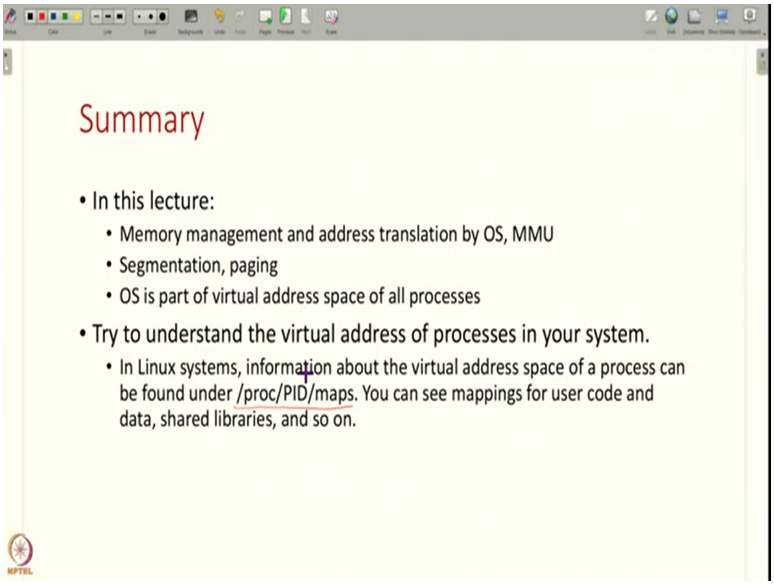


Then the question might come up with the process can access OS code and data how are we protecting from illegal access? Can the user go and change the OS code and data. Well, here is where the page table has permissions. So, page table for every page you will store a permission saying you know can you read is it only a read only page or can you also write to it is it user page is it kernel page all of these permissions will be stored so that a user program only when it

is in kernel mode, when it jumps into kernel mode after a trap instruction only then it can access this OS code and data otherwise it can only access it user code and data.

So, you cannot just randomly jump to the high virtual addresses of the OS code, the OS code even though it is located at certain virtual addresses in your page table, you cannot access it unless there is a trap. So, these permission checks are also done by the MMU. Whenever you request if you say you want to request some OS code or data at a high virtual address and you are in user mode, the MMU will consider this as illegal access and it will track to the operating system. All of these permission checks like, is the page read only, can you read and write to it all of these are also checked by the MMU.

(Refer Slide Time: 29:12)



So, that is all we have for today's lecture. In this lecture, we have studied how the operating system allocates memory for processes and manages the address translation along with the MMU. We have seen concepts like segmentation and paging. And we also understood that the virtual address space of a process not just has the code and data of the process, but also other common pieces of software like shared libraries or operating systems that the process will need to access from time to time. These are all part of the address space of a process.

So now, you can try to understand this concept of virtual addresses and virtual address spaces in your system. In Linux machines for example, you can look up files under slash proc in this proc file system directory you can see what are the virtual addresses assigned to a processes code,

stack, heap to various libraries all of these this information you can see to understand this concept better. So that is all I have for this lecture. In the next lecture, we want to understand paging and a little bit more detail. Thank you all.