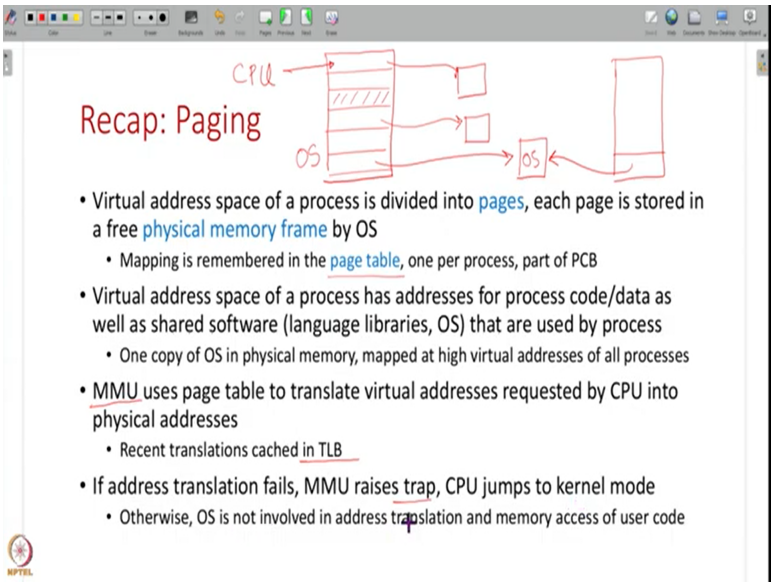
Design and Engineering of Computer Systems
Professor. Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 12
Paging

Hello everyone, welcome to the twelfth lecture in the course design and engineering of computer systems. In this lecture, we will continue our discussion of memory management and we will go into more detail on the concept of paging that was introduced in the previous lecture. So, let us get started.

(Refer Slide Time: 00:32)



So, this is a recap of what we have seen in the previous lecture, every process has a virtual address space. It is all the set of virtual addresses that the process can access and this is divided into fixed size chunks called pages. And each page is assigned a memory frame in RAM by the operating system, a free physical memory frame is assigned to every logical page to store its contents in RAM. And this mapping of which page is stored at which location in RAM this is remembered by the page table.

Now this virtual address space, we have seen that it not only has the process code data, stack, heap, it also has some shared software like the C library or the operating system. So, shared software that is used by multiple processes that is also part of the virtual address space of every process. For example, the high virtual addresses that are not used by the user code are used by

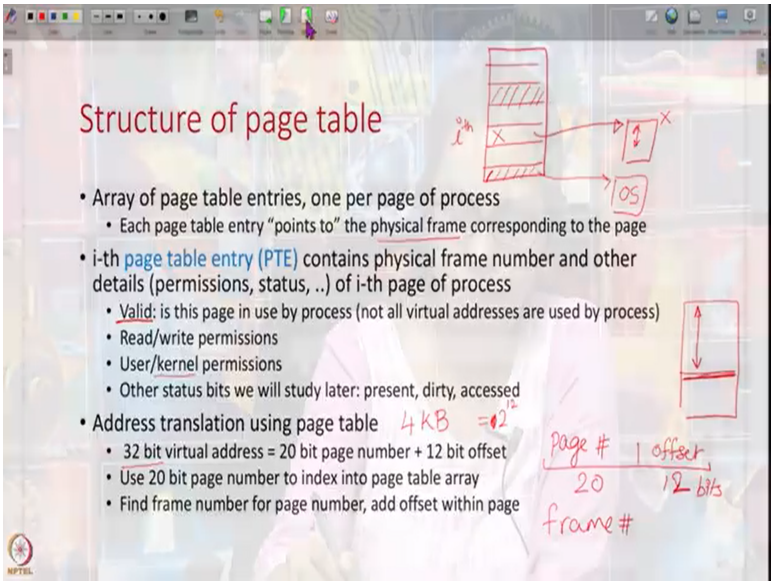
the operating system. And the page table maps these high virtual addresses to the physical location of the OS code in RAM.

And note that there is only one copy of the operating system in RAM. Another process also its page table also will map these high virtual addresses to the same physical addresses in RAM. So, with shared software, the same physical frame can appear in the virtual address space of multiple processes by adding these mappings into the page tables of these processes.

And once this page table is created by the OS, the MMU a special piece of hardware called the MMU uses this page table uses these mappings from virtual addresses to physical addresses to translate virtual addresses. So, whenever the CPU requests code or data at some virtual address, this translation is used by the MMU to get the physical address and the recent translations made by the MMU are cached in the TLB.

So, if the address translation fails for some reason, for example, some virtual addresses are not in use by the process, but still, the process accesses them in such cases, if the address translation fails due to some error condition, then the MMU raises a trap and the CPU jumps to kernel mode we have seen how these traps work and the operating system will solve this problem. So, this is a recap of what we have seen in the previous lecture with respect to paging. So, now, let us understand what is the structure of the page table.

(Refer Slide Time: 03:18)



You can think of the page table as a large array of page table entries. If the process has n pages, there will be n entries in this array and each page table entry will contain the physical frame number, where that page is stored. So for example, the ith page in the process, if it is stored at some physical frame number say x, then the ith page table entry will have this physical frame number x stored in it, so the ith page table entry contains the frame number and some other information of the ith page of the process. So, this is an array with one entry for every page of the process.

So, we also indicated colloquially like a pointer because it contains the frame number. We also show it as a pointer to that physical frame. So, in addition to this physical frame number in addition to this pointer to physical memory, what does a page table entry contain? It has some other information about the page like for example, is the page table entry valid or not? That is, some virtual addresses may not be used by the process. In a 32-bit architecture, you have 4 GB of virtual address space, you may not have 4 GB of code or data to store so some virtual addresses you may not use.

So, such virtual addresses such pages are marked as invalid, the page table entry does not have this valid bit set that means you will not store any frame number corresponding to that page because that page is not in use. So, that is one piece of information that is stored in a page table entry. Other things are like Read, Write permissions user kernel permissions. For example, if this page if these virtual addresses are mapping to the physical location of the OS code, the high virtual addresses on the page table map them to the physical locations of the OS Code, we have seen this.

For such page table entries, which contain a pointer to OS code, you will set a permission bit indicating that they should only be used in kernel mode and not in user mode. So, such permissions are stored with every page table entry indicating how the page should be used. So, in addition to these permission bits, there are also a few other status bits about every page that we will study in the next lecture.

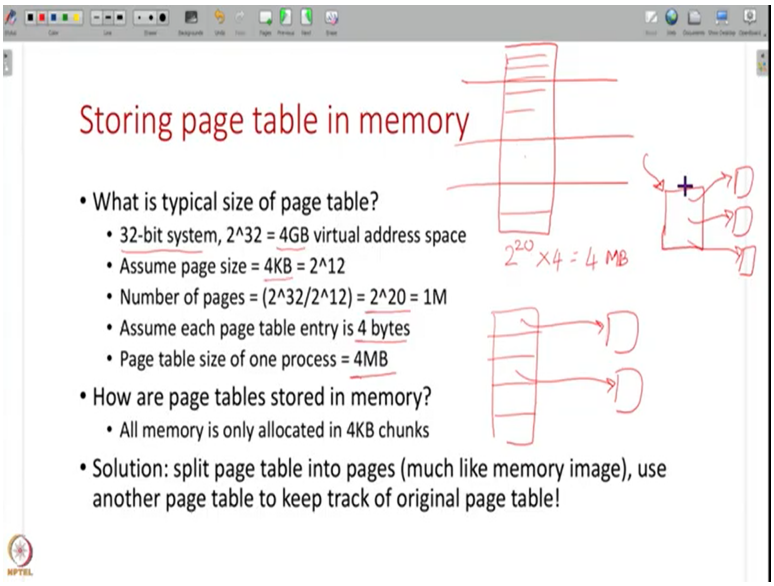
So now, you have this page table with one entry for every page, then how do you translate a virtual address to a physical address using this page table? So, let us consider 32-bit architectures and assume we have 4 KB pages, which is the standard today. So now, within a page, you have 4 kilo bytes. So, 4k is nothing but 2 to the 12.

So now, you have 2 to the 12 bytes in this page, so 12 bits in your virtual address will indicate the offset within this page, so the last 12 bits will tell you which of these 2 to the 12 bytes in a page are you referring to. And the remaining 20 bits will tell you the page number. So, how do you translate an address using a page table, you will take a 32 bit address, and split it into a page number and an offset.

And this page number, you will look up in the page table, you will go to whatever is the page number, you will go to that entry, you will use this page number to index into the page table, find the corresponding page table entry, read the frame number from it, and then replace the page number with the frame number and along with the offset, because in that frame, also the same offset will be used, the offset does not change. If the pages and frames are the same size whatever is the offset here will be the offset here also. Use the same offset replace the page number with the actual physical frame number and the combination of these two will give you the physical address.

So, this is how you translate virtual addresses to physical addresses using this page table array. So, now let us go into a little bit more detail of what is the size of this array and where is this array stored in memory. We have said that this page table is part of the PCB of a process. So, it must be somewhere in the OS, part of the memory. The OS has its own code and data structures. The PCB is also one such data structure that the OS maintains and as part of that PCB, you can store this page table also. But now let us see what is the size of this page table to think about the feasibility of how it can be stored in memory.

(Refer Slide Time: 08:31)



So, what is the typical size of a page table, if you have a 32 bit system, then you have 2 to the 32 that is 4 GB of virtual addresses available to every process. And if your page sizes 4 KB that is 2 to the 12 bytes, then the number of pages will be 2 to the 32 divided by 2 to the 12 that is 2 to the 20 almost a million pages you have in the virtual address space of every process.

A process may not use all these million pages all the time, but that many addresses are available for every process. And so your page table is an array of 2 to the 20 entries, and all of these entries have to be maintained whether the process is using the memory or not whether they are valid or not. If the entry is not valid, you will not set the valid bit but all these 2 to the 20 entries will be maintained in the page table of every process.

And if each entry is 4 bytes in size, then your entire page table size will be 2 to the 20 into 4 that is 4 MB. So, you have 2 to the 20 entries each have 4 bytes. So, the total size of this page table of processes 4 MB this is quite huge. If you have a small program, it still needs 4 MB of page table memory to keep track of the program state. So, now the question comes up how do you store this page table in memory such a large chunk.

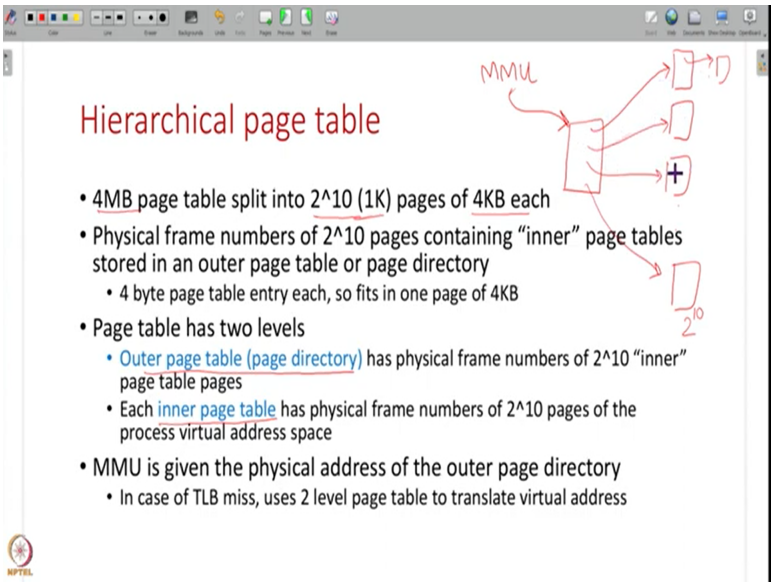
Note that RAM is only allocated in page size chunks you only allocate 4 KB at a time. So, how will you store this 4 MB of data. The way we do it is we will split this page table also into 4 KB chunks, this page table also will be split into smaller pages and distributed throughout memory. It is the same concept, there is the entire memory image of a process just like this memory image is

split into smaller chunks and stored at different locations in RAM. Similarly, the page table also will be split into smaller chunks and stored at different locations at different frames in RAM. That is the only way memory allocation can be done in paid sized chunks.

Now, if your page table is split into multiple such locations, then how do you tell the MMU where is the page tables. You remember that we tell the MMU here is a pointer to the page table using which it will translate. Now if you split your page table into so many chunks then which chunk do you give the MMU.

So, what we do is we will use another page table we will use another outer page table to keep track of the physical locations of these page table chunks and we will give the MMU a pointer to this outer page table. So, we have to do this we cannot give the MMU pointer to like the zillion chunks and say oh go read the page table from here. So, therefore, we will collect the physical locations of all these page table chunks and put it in another table and give the MMU a pointer to this table.

(Refer Slide Time: 11:42)



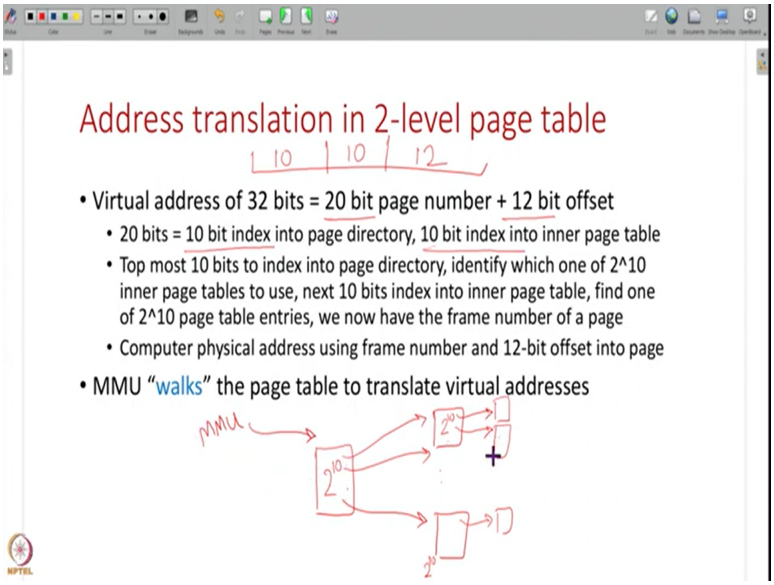
So, this concept is called hierarchical paging and such page tables are called hierarchical page tables. So, in the previous example, this 4 MB page table, we will split it into chunks of 4KB each and how mean such chunks will be there, you will have 1k or 2 to the 10 such chunks. So, your page table is split into so each of these chunks has page table entries has some page table entries, and you will have 2 to the 10 such inner pages which have page table entries.

Now, these page table entries point to the physical frames of the process. And then you will have an outer page table which has the physical frame numbers have all of these inner page table chunks. And now this 2 to the 10 frame numbers of this inner page table each assuming it is 4 bytes again like a page table entry it will fit in 4 KB.

So, one page will be enough to store all of this information and this pointer to this outer page table is what you will give the MMU. So, this is called a two level page table you have an outer page table also called the page directory, what does this have, this has the physical frame numbers of these 2 to the 10 inner page tables. And each of these inner page tables each has 2 to the 10 frame numbers of the memory pages the physical frames of your process. So, this is your actual page table array that is split and these individual chunks are kept track of in this outer page table you have an outer page table and multiple inner page tables.

And now when the MMU is given a pointer to this outer page table or the page directory, and if there is a TLB miss then the MMU has to look up this two level page table to translate a virtual address to a physical address. That will that is it will first find out which of these inner page tables to use, and then inside that it will find out the page table entry of the virtual address lookup the physical frame and then translate. So, let us see how that is done, how address translation is done.

(Refer Slide Time: 14:03)

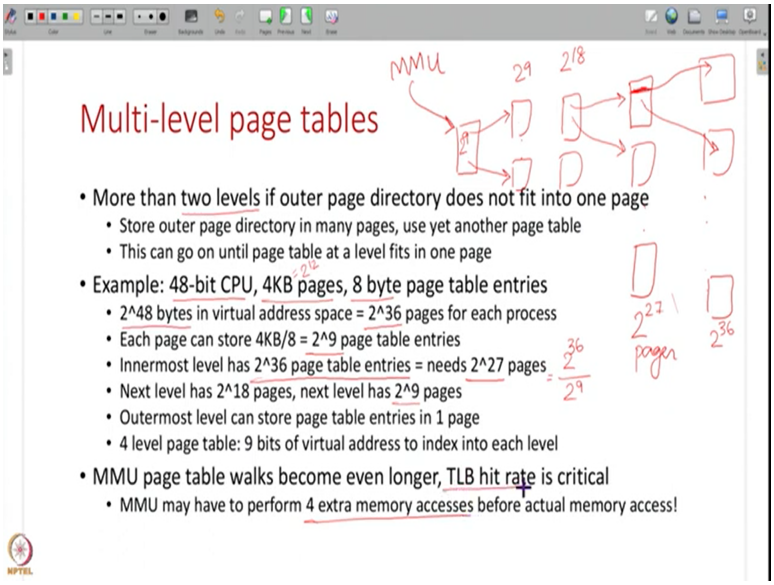


So, you have 2 to the 10 inner page tables each of which has 2 to the 10 page table entries which contain the physical frame numbers of the actual memory pages of a process where the process code data stack heap everything is stored. And there is an outer page table which has 2 to the 10 physical frame numbers pointers to all of these inner page tables and the starting address of this outer page table is given to the MMU. So then, your virtual address your 32-bit virtual address you have split it into a 20 bit page number and a 12 bit offset.

Now this 20 bit page number you will once again split it into a 10 bit index into the page directory and a 10 bit index into the inner page table. So, to locate your page in this array you will first see which of these chunks of this array should I go to you will look at the most significant the most significant 10 bits, you will see to find out which page table chunk should I go to. Inside that page table chunk you will use the next 10 bits to index into one of these 2 to the 10 page table entries in the inner page table.

And now you have obtained your frame number then you will add the 12 bit offset and you will get your physical address. So, this is called MMU is walking the page table. If your page table has multiple pieces, then the MMU will first look up, read this page, find something in it, then go to this page, find something in it and then it will get the physical address to access the actual memory of your process. This process of traversing this hierarchical page table is called Walking the page table.

(Refer Slide Time: 16:06)



Now, you might ask if there are multiple such inner page tables and you know, you have put it in one outer page table. What if it does not fit in one outer page directory? Then what do you do if you have multiple such outer page directories then you will have to have another page table to keep track of all these page directories. And you have to keep doing this until at some point all the information fits in one page and that one page can be given to the MMU the address of that one page can be given to the MMU. So, you can have more than two levels in the outer page directory if required.

So, let us see an example. Suppose you have a 48 bit CPU that is your virtual address space is 2 to the 48 bytes in size. And you have 4 KB pages and 8 byte page table entries. So suppose this is your situation, the previous 32 bit example was simpler. Now, we will see that this will get more complex.

So, now what do you have, your process has 2 to the 48 bytes in its address space, divided by 4 KB, this is 2 to the 12 bytes. So, if you divide this, you will get 2 to the 36 pages are there, possibly not every process will have 2 to the 36 pages, but a process can have up to 2 to the 36 pages. And therefore, you will have 2 to the 36 page table entries in the page table of the process.

Now, every page every 4 KB page can store 2 to the 9 page table entries 2 to the 12 divided by 8 that is it can store two to the nine page table entries. So how many, what is the size of the innermost page table of a process, you have 2 to the 36 total page table entries divided by each page can store 2 to the 9 page table entries.

So, you will need 2 to the 27 pages to store all the page table entries of the process in the innermost level, you have 2 to the 27 pages in the innermost level of the page table of the process. And these 2 to the 27 pages will contain the frame numbers of these 2 to the 36 pages of the process. So, you have 2 to the 27 pages in your innermost level of the page table. Now these 2 to the 27 pages, their frame numbers have to be kept track off in a page directory.

And how many such pages do you need all these 2 to the 27 will not fit in one page anymore, because a page can only hold 2 to the 9 page table entries. So therefore, you will have 2 to the 27 divided by 2 to the 9 which is 2 to the 18, you have 2 to the 18 pages in the next level have the page table which contains pointers to this inner most page table.

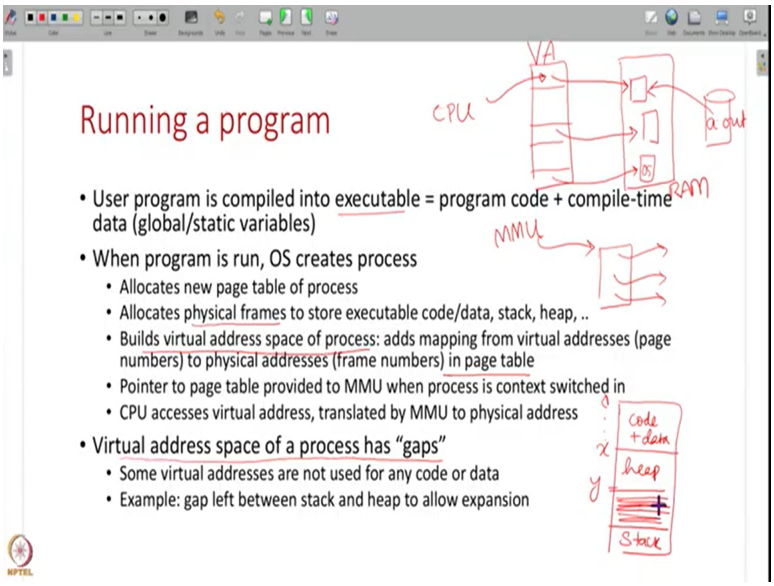
Then in the next level, you have 2 to the 9 pages. And all of these frame numbers can fit in one page and this page will be given to the MMU for address translation. So, you have an outer most fourth level outer most page directory, this has 2 to the 9 frame numbers to 2 to the 9 pages. These 2 to the 9 pages, each in turn has pointers to 2 to the 9 pages. So, you have 2 to the 18 pages. These 2 to the 18 pages contain pointers to these 2 to the 27 pages.

And this 2 to the 27 pages have the 2 to the 36 page table entries, the 2 to the 36 page table entries required for this process. So, you have a 1, 2, 3, 4; 4 level page table. And how do you translate a virtual address to a physical address you will first take the first topmost 9 bits index into these 2 to the 9 find one of the entries go here next 9 bits next 9 bits next 9 bits by using this 36 bits. The first 36 bits you have located the final page table entry corresponding to this page, then here you will find the actual frame number and you will translate the address.

So, now to translate one virtual address to physical address, the MMU has to perform 1, 2, 3, 4 extra memory accesses before it can obtain the physical address using which it can actually access the memory contents. So, before every memory access, there are four extra memory accesses if there is a TLB miss, which is why once you have multi level page tables, in you know more than 32 bit CPUs, if you have a 64 bit CPU, you will have multiple levels of the page table like this. In such cases, a TLB hit is even more important, otherwise, you are going to have many more extra memory accesses. So, this is about multi-level page tables.

So, now let us take a step back we have understood how the operating system manages the virtual address space of a process divides it into pages places them at different physical memory frames and keeps track of this mapping in the page table. So, this is the view from the operating system side. Now let us come out of the operating system and look at things from a user point of view now you as a user you have written your program. So, from your point of view, what is happening how is all this memory allocation happening.

(Refer Slide Time: 21:45)



So, you have written a program you have compiled it into an executable, this executable contains the code as well as the compile time data like global, static variables and so on. So, now this a dot out that you have compiled is say on disk somewhere and when you run this program, the OS creates a process. So, it will assign some memory frames in RAM, it will find some free physical frames in RAM.

So, this is your physical memory, it will find these free physical frames, and it will copy the content from the executable into these frames and some frames can also be just free pages for the stack heap and so on. So, this physical memory will be allocated and then it builds your virtual address space that is at these virtual addresses starting from 0, the code and data is mapped that is you will have add a page table entry from this page to this frame, then you will have another page table entry to the stack to the heap.

And also if your OS code is located somewhere you will add a page table entry from the OS virtual addresses to the OS code in RAM. So, it will the OS will build the virtual address space of a process. That is it will create a page table and add all of these page table entries. The frame number which has the code and data will be stored in the first page table entry the frame number that has the stack will be stored in some page table entry.

The frame number that has the OS code and data will be stored in some page table entry corresponding to the OS virtual addresses. In this way, this page table is constructed by the

operating system and by constructing this page table the operating system has constructed your virtual address space it has constructed what are the virtual addresses and what is the program's view of the memory the OS has constructed. It is giving this illusion that this is the memory you have by means of this page table.

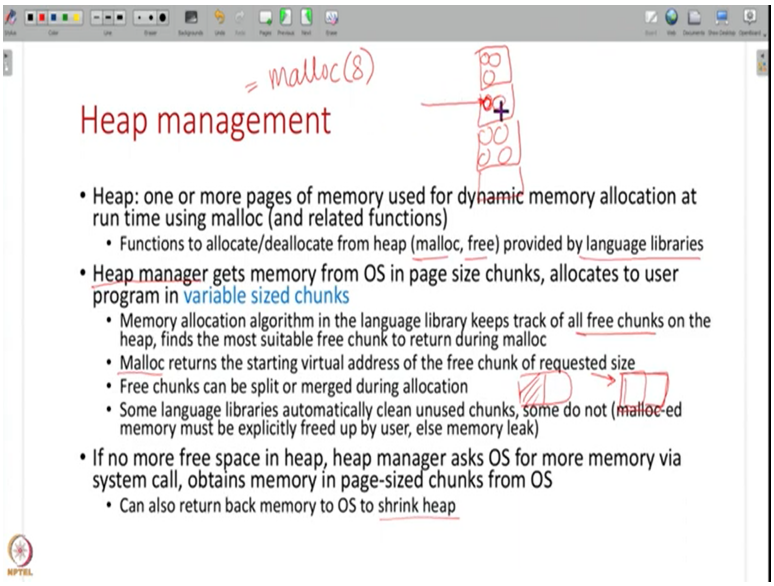
And then this page table's pointer is given to the MMU when the processes context switched in, so the OS creates a page table during say the fork system call when the process is created the page table is created, but the MMU is told about it only when the process runs when it is context switched in.

And then when the process is running the CPU will access some virtual address program counter will say is pointing to this code over here the CPU will try to fetch this instruction and then the page table will be used to translate it into this physical address and the actual memory will be accessed. So, this is how the OS creates a process and you as a user will have this view of the virtual address space you will see code data being placed at these different virtual addresses and so on.

And, whenever you look at your virtual address space, you will see that it has some gaps, not all virtual addresses will be in use for example, the OS might allocate say virtual addresses 0 to some number x for you know the code plus the compile time data. Then from x to y it can allocate for the heap and then it can leave some addresses free and then allocate the stack over here. These virtual addresses will not be assigned to anything there will be such gaps in your virtual address space. And if you access this virtual address, it will throw a trap.

So, in the page table, these entries for these pages that correspond to these virtual addresses will be invalid. Now, why does the OS do that? Why cannot you just put everything one after the other because you want the heap may expand the stack may expand? So, you want to leave gaps to allow for expansion for example, if your heap needs more memory, it will start to use these new virtual addresses that is why you will have gaps in your virtual address space.

(Refer Slide Time: 25:56)



And a little bit more detail on the heap. So, the heap is basically one or more pages of memory that your C library or some language library the heaps are managed by the language library in C language, it will be the C language library and so on. So, these language libraries will get one or more pages from the OS and they will allocate smaller chunks to user programs. If you as a in your user program if you say MALLOC, you want say 8 bytes, then a small 8 by chunk will be allocated on this heap memory and return to you.

So, MALLOC will basically return the starting address starting virtual address of this freeze chunk. And inside this heap will have you know some chunks are unused or unused by the process some chunks the process has freed up whenever you do MALLOC later on you will do free to free up that memory.

So, this heap manager the process the code that manages the heap will split this pages into smaller size chunks and will allocate it to the user and it will keep track of all of these free chunks. So, suppose you have allocated this chunk and then it has been freed up then later on when somebody else requests it, it may give this chunk. So, this free chunks if you have a bigger free chunk it can be split and a smaller one can be allocated or if you have two adjust and free chunks they can both be merged and this bigger free chunk can be returned to the program.

So, you will manage these free chunks using some suitable algorithms and these smaller chunks will be returned to the user program when you call functions like MALLOC. So, this MALLOC

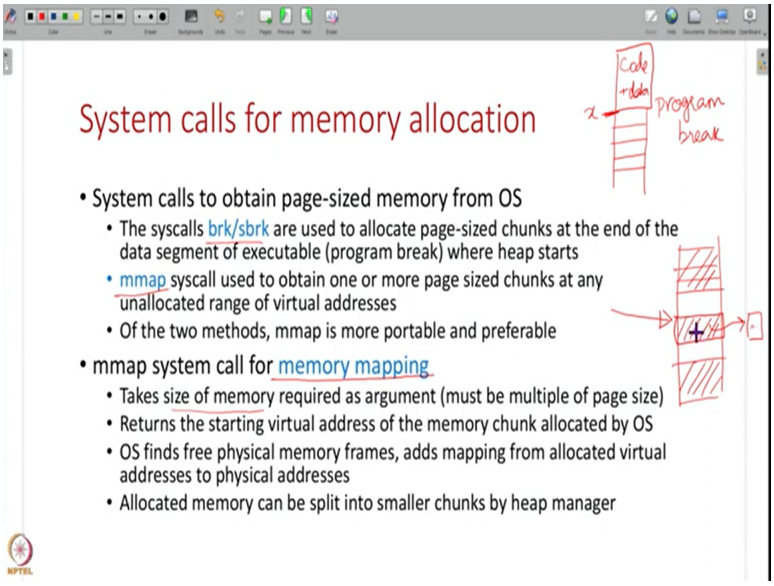
the memory that MALLOC is returning is actually obtained in page size chunks from the OS why because always will only give you page size chunks it will not give you 8 bytes and these pages are split into the smaller chunks and given out by the heap manager program.

And some heap managers some language libraries automatically clean up if you have done MALLOC and you are not using this memory anymore, they will automatically clean it up seeing that there are no pointers to this memory anymore that is called you can you know reference count the pointers to this chunk and free them up that is called automatic garbage collection. Whereas, in some languages like C you have to explicitly free up if you do MALLOC you get a pointer, then you have to call free on this pointer to free up this chunk.

And some heap managers who can also you know return back memory to shrink the heap. And if you are heap is out of memory you have allocated smaller chunks to the program and all your pages are full, then you can also request for more memory from the operating system. You can request for more memory you can give back existing memory.

So, the heap deals with the OS and page size chunks in this way and the user program deals with the heap using functions like MALLOC to get smaller chunks. So now, how does you know the C language library the heap manager program if it wants more memory from the OS, how does it ask for more memory.

(Refer Slide Time: 29:12)



So, there are a few system calls for that. So, one system call is called the break or the SBRK/BRK or SBRK system call which is it allocates memory from the program break that is your code plus compile time data is there in your virtual address space say up to some address x then the BRK/SBRK system calls are used to allocate additional memory at from this point onwards after the code plus data section. This point this virtual address is called the program break. And from this point onwards, if you want to allocate some pages of memory you use the system calls.

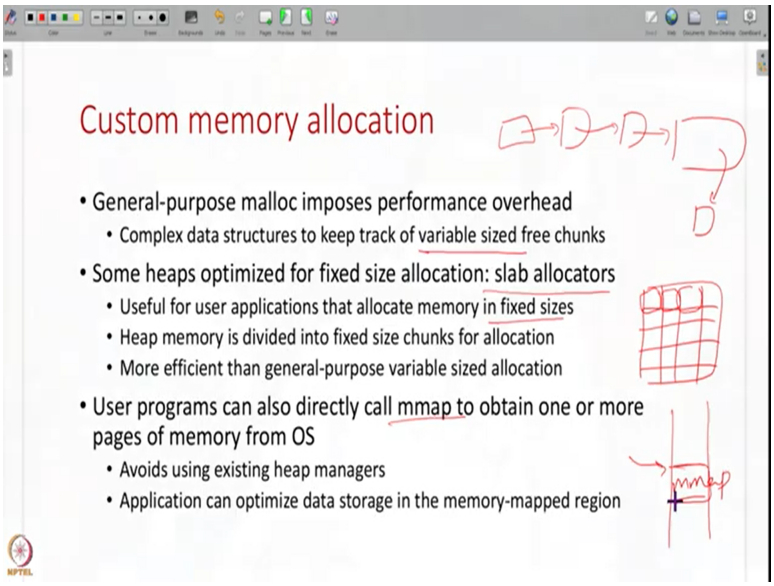
On the other hand, there is another system called mmap that can be used to allocate memory at any virtual address. So, in your virtual address space you have the Code Data some virtual addresses are in use. And some other virtual addresses, say here is the stack, then any free virtual address range that is not in use yet, you can say, allocate a physical frame for me and map it to this page number, you can use the mmap system called to do that you can get a page size chunk at any free virtual address that is not yet already in use.

So, the BRK/SBRK system calls allocate memory at specific virtual addresses after the program break the mmap system call allocates memory at any unallocated virtual address, there will be many such gaps in your virtual address space. But either of these, you cannot ask the OS for 4 bytes 8 bytes, it will only give you 4 KB or whatever is the page size on your system only that size chunks you will get.

So, this mmap system call is very powerful it is it stands for memory mapping, it takes as an argument the size of the memory that is required, it should be some multiple of page size, and it will return when you do mmap you will get the starting virtual address of this chunk. And internally, the OS has allocated a free frame and added a page table mapping from this frame number to this page number in your page table so that when you access this virtual address you are actually accessing this physical memory allocated to.

So, the heap can also for example, do a map get chunks like this, and then split this into smaller chunks and give it to the user program. So, this is one way of expanding your heap. Now, this general purpose MALLOC, the heap manager that you have by default usually has some performance overheads.

(Refer Slide Time: 31:50)



Why because, there are you know these variable size chunks that are all maintained and say some link list or something, there is a big chunk, there is a small chunk, there is a list of free chunks, whenever you want some memory allocation you will go through this list you will find something oh this is too small for me, this is too big for me.

So, allocating memory from these list of free chunks, you know this variable sized allocation usually is a little bit more of a headache, it has some higher overhead. So therefore, what some heaps do is they will allocate memory in fixed size chunks. So, such heap managers are called slab allocators, they will divide your heap into you know like a grid into fixed size chunks and they will give you a standard size every time.

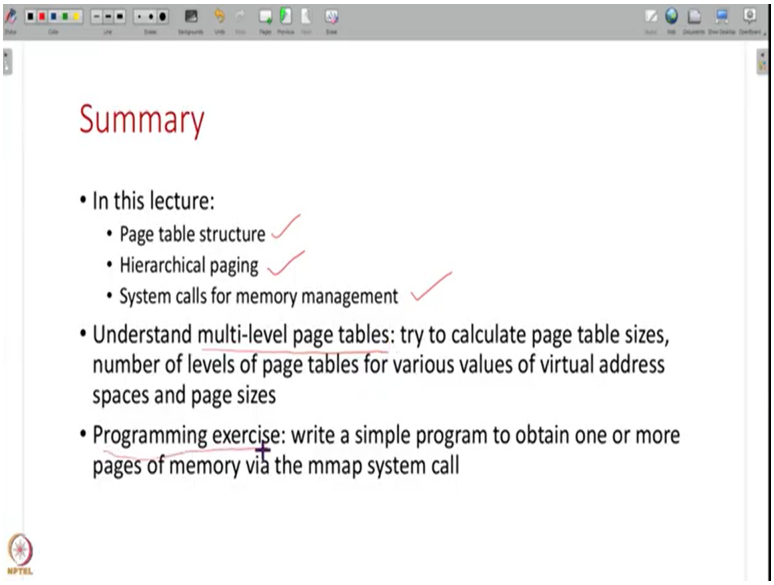
So, this is not for every program. If your application has this characteristic that it only allocates memory in certain fixed sizes. If your application logic is such that you will only allocate memory in either 8-byte chunks or 16-byte chunks or something like that, if you know this a priori, then you can use an optimized memory allocation algorithm in your heap called a slab allocator.

Where you divide it into fixed size chunks and you will allocate these fixed sized chunks. So, this is in general faster if you do MALLOC like this with a slab allocator it will only give you fixed sizes, you don't have flexibility, but it will return faster you don't have to traverse like some variable sized chunks and all of that.

So, this is one optimization you can use in your programs if your application has this property. And the other thing that you can do is if you want faster memory allocation, you can directly bypass the heap altogether and directly use mmap to obtain memory in your program you can say okay, there is some heap stack whatever, but I want to store large amounts of data.

So, I will directly obtain in my virtual address space somewhere I will you know mmap some memory and in this memory, I will put my own data and you can optimize your data storage by memory mapping yourself and getting memory directly from the operating system. So, this is for advanced applications that have to store a large amount of data. So, that is all I have for this lecture.

(Refer Slide Time: 33:59)



In this lecture, we have gone into more detail about the page table structure and hierarchical paging. And we have also seen what are the system calls involved for memory management how the user program or other libraries like the heap manager can get extra memory from the OS using system calls like mmap and SBRK. So, an exercise for you is understand this concept of multi-level page tables, you can try to calculate how many levels are there in the page table for different architectures for different page sizes and so on.

And a small programming exercises you know, try to write a simple program using the mmap system called the mmap system call is very powerful and very useful in real systems. So, try to write a program that allocates a chunk of memory you can put some data in that memory region

you have allocated and play around with it. So, thank you all that is all for this lecture. We will continue this topic of memory management in the next lecture. Thanks.