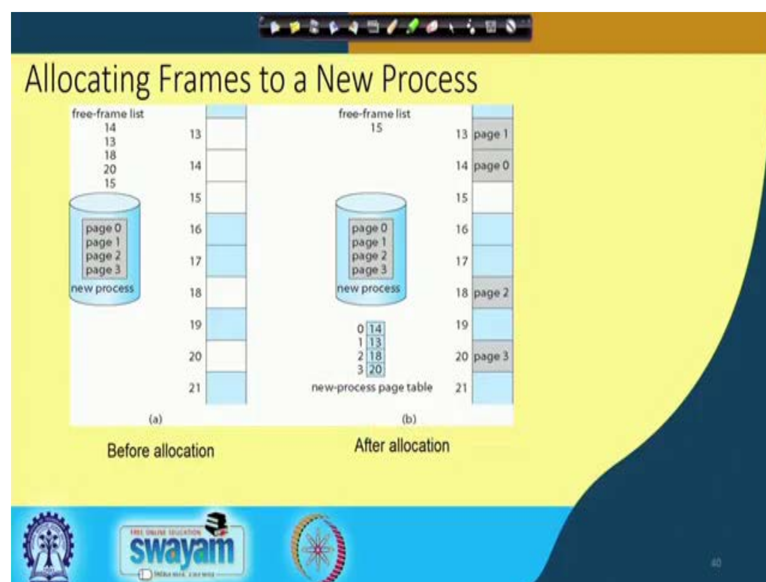
Operating System Fundamentals
Prof. Santanu Chattopadhyay
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture - 46
Memory Management (Contd.)

Next will be looking into the situation like as the frames are allocated to pages of a new process how this situation of the memory changes ok.

(Refer Slide Time: 00:34)



So, in this particular example that we have supposed this is the current scenario, so blocks which are marked in blue. So, they are the they are used and this others are free; like, we have got this 13 14 15 then this 18 20, so these are the free frames. Now, apparently it seems why do not we have these free frame list like this 13 14 15 18 20, but rather 14 13 18 20 15 like this.

So, this can happen because may be at some point of time this block 14 this frame 14 was occupied by some process and that released that ended and release that particular frame as a result it came into the free frame list. After that may be the frame 13 was released by another process, so this 13 came into the free frame list.

So, that is how this may be the frame were released by different processes in this sequence as a result this free frame list came like this. So, since it is not mandatory that

this processes be allocated frames contiguously, so the it is not necessary to keep this free frame list sorted ok. So, if you keep them sorted then of course, the every time a new frame is added, so you have to run some sorting algorithm change the structure of this frame list, so that is an overhead.

So, it is simply not done, so it is kept in that whatever order this frames are being freed, so it is kept in that order only. Now, this suppose a new process has arrived which has requirement of 4 pages page 0 1 2 3. Now, this page 0 1 2 3, so the process will be allocated space and suppose after allocation, so the it will allocate this page 0 to frame 14 then the next free frame is 13.

So, page 1 goes to 13, page 2 goes to 18, and page 3 goes to 20, so that is how they have been allocated. So, page 1 page 0 is at 14, page 1 at 13, page 2 at 18, and page 3 at 20. And we have got this page table for the process created. So, 14 13 18 20 for the indices corresponding to the indices of this page table, so they have been. Now, the free frame list it has got only one free frame now which is 15. So, this is the this is how this frames are allocated to the to a new process as the new processes are coming.

(Refer Slide Time: 03:02)

Now, the difficulty with this paging scheme is what we were discussing previously that to access once the one particular memory location we are accessing the memory twice. Because the page table is also a part of memory, because page table can be very big like

the individual processes can be quite large and then if a processes has got large number of pages then it is page table will is also very large.

So, there is no way to keep this page table in the CPU registers and all, so we have to keep the page table in the main memory only. At the same time we cannot keep this page table in the secondary storage or disk, because in that case to access one primary memory location, so we have to make a second one secondary storage access. And secondary storage access is even slower much slower than the main memory access. So, that way this whole scheme will fail.

So, what is done? Is that this memory if you look into, so you can think that as if this whole memory is divided into two parts in one part of the memory along with the ways and all. So, we have got some portion where all the page tables are kept, so these are the page table area you can say.

And then we have got the remaining part which is actually divided into pages and these pages are going to be accessed the processes are going to be allocated space from this page table. Now, this page table area, so that is also a part of memory. So, whenever you are trying to access one particular memory location first you are accessing this page table area to find out the corresponding frame number and after getting the frame number we are coming to the corresponding frame and accessing the corresponding location.

So, that is why one memory access is getting translated into two memory accesses which make the overall computation by the processor slow. So, if the page table is kept in memory every data or instruction access requires two memory accesses one for the page table, and one for the data or instruction. The two memory access problem can be partially solved by the use of a special fast look up hardware cache called associative memory or Translation Look aside Buffers or TLBs.

So, what happens is that between CPU and memory, so we have got this CPU. So, it is generating the logical address consisting of the page number and offset and that page number it is trying to search into this page table to get the corresponding frame number ok. To get the corresponding frame number and this displacement will come and then it will be accessing here.

Now, to reduce the time to access this page table area what is done between this CPU and memory. So, we have got a cache and in this in this cache, so we are keeping a part of the page table entire page table may not be possible to be kept in the cache because of the size limitations of the cache. So, we keep a part of the page table into this cache which we called translation look aside buffer.

So, this page table is searched simultaneously into this cache ok. So, if it is found that it is it is there then the corresponding frame number is there then it will be used from here directly. So, that way since cache is going to be much faster than the main memory, so this two memory access time. So, at least we can save 80 percent time of the first memory access, so that can be this done.

And this associative memory it has got parallel search mechanism. So, given a page number, so it can search each and every location parallelly. So, so given this value of P, so it will search for P in all these locations and that check is done parallelly. So, that is why it is fast, and then if it finds any match then the corresponding frame number is retrieved and returned to the CPU that this is the a value.

So, this address translation mechanism, so if p is in associative register get the frame number out, otherwise get frame number from the page table in memory. So, that is the idea and if you, so this if otherwise means, so this is also called a cache miss situation. So, this is called a cache miss situation.

So, if a cache miss occurs, in that case it has to access page table and get the frame number from there. And while that is done, so this cache is updated with that page number frame number combination, so that in near future if the same page is accessed then we do not need to go to the page table again.

So, it can just take the data from the cache the frame number can be obtained from there. So, this is how this TLB based this page table operates, and this the TLB is there then the access is much faster than this main memory based policy.

(Refer Slide Time: 08:14)

TLB issues

- TLB is typically small (64 to 1,024 entries)
- On a TLB miss, the value of the (missed page-table and frame-number), is loaded into the TLB for faster access next time that address is used.
  - What if there is no free TLB entry? Replacement policies must be considered
  - Some entries can be wired down for permanent fast access
- Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush TLB at every context switch

So, the issues that we have is that TLB is typically small, because of this cost of the system or cost of the TLB, so we cannot have a large TLB; so 64 to 1024 entries can be kept in the TLB. So, hm, so TLB size is a small is a problem, so if the process has got large number of pages, so we cannot accommodate all the page entries in the TLB.

More on a TLB miss the value of the missed page table and frame number is loaded into the TLB for faster access next time that address is used. So, actually if you look into the programs behavior then this programs they have got the strategy they have got the tendency to have something called the locality of reference.

So, locality of reference means that if you have accessed one particular instruction at this point of time, so it is very much likely that in near future you will be accessing in the vicinity of this particular instruction ok. Because, programs generally execute in a straight line at in an analysis there is a jump or some error condition like that. So, that way it is, so if there is a cache miss here; then when that cache miss occurs then for the next. So, if you load this cache with the page number, frame number combination from main memory page table into it

So, for the next few instruction, so it is very much likely that there will be no cache miss; so that way it will speed up. So, this is one thing and this is called the locality of reference and if there is a TLB miss the value of the missed page table and frame number is loaded into TLB. For faster access next time that address is used, or the addresses

which are in the vicinity is used. What if there is no TLB, free TLB entry? So, then replacement policies must be considered.

So, it may so happen that TLB has got only say 64 lines and they are, so may be at present in the TLB there are 60 entries have been loaded. So, still there are 4 more entries which are free, so these are not used at present. So, in this particular case when a miss occurs, so the corresponding frame number can be loaded into this one of these free slots.

However, if all of them are used, so if all the 64 have been loaded previously, then we have to think about some replacement. Like we have there are some cache replacement, policy that tells whichever entry has not been used in recent time, so that has to be replaced and all, so that is there

So, some entries can be wired down for permanent fast access; maybe there are some routines, so which are very frequently required by the process. So, we can say that the entries for page numbers say 1 5 and 9, so they are very important, so they should never be replaced. So, that is there, so that is called wired down, so some entries can be wired down for permanent fast access. Some TLBs store address space identifiers or ASIDs in each TLB entry that uniquely identifies each process to provide address space protection for that process.

So, that, so for some protection mechanism, so it will be like who which process has created it and all, so that way it can be useful for giving the protection. Otherwise, need to flush TLB at every context switch. So, whenever a process changes, so a process makes a transition from running state back to ready state it goes back to the ready state.

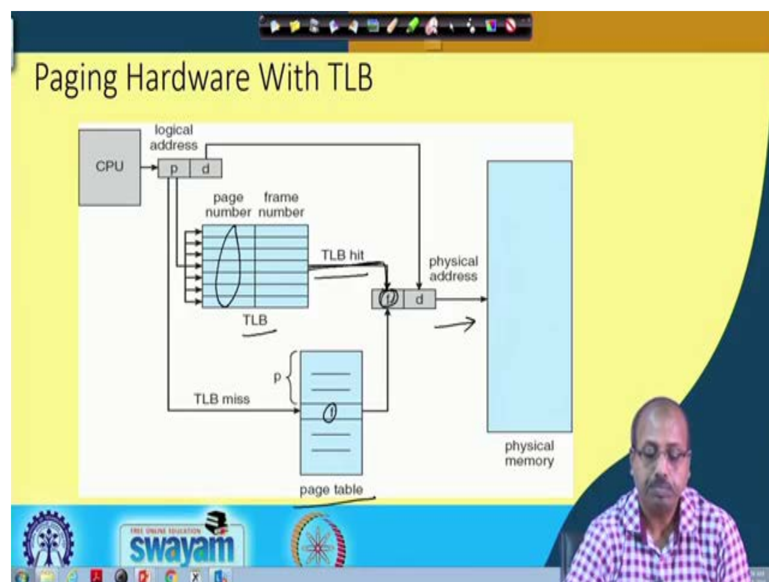
So, another process will go from ready to run state, so this is the context switch. So, whenever this context switching is occurring, so this TLB becomes invalid the TLB context become invalid. So, this as a result this TLB content has to be flushed out and that TLB has to be loaded with the page table entries for the next process to run.

So, one policy may be that we do not do anything, so next time the TLB is entry now and whenever this next process starts. So, initially it will be facing some TLB misses, but accordingly from the page table the TLB will get populated. So, after sometime so it will become stable, so this is one possibility.

Another possibility is that you can whenever you are loading pages whenever you are loading the program or starting the program. So, on the from the page table the first few entries can be copied into the TLB, so that is also possible.

So, this address space identifiers, so this actually some TLB entries are kept bound to some processes, so this, so that other processes they will not be flushed out. So, it will tell which process created which process is using this particular page. So, as a result, so they will not be flushed out; otherwise they will be flushed out at every context switch, so this is there.

(Refer Slide Time: 13:15)



So, with the with the TLB being present the address translation mechanism will be something like this or paging hardware it will work like this. So, the CPU it generates the logical address in terms of this page number and displacement; and this page number it is given simultaneously to the TLB as well as page table. So, it goes to the TLB search and the main memory search, so it goes to both of them simultaneously.

Now, in the TLB search so it will be finding this page number frame number combination. So, if this page number is found in one of these entries here, then we are fortunate, so then this a TLB hit occurs; in that case this frame number comes from here. If there is a TLB miss that is it could not find the entry here, so in that case it will from the page table it will find out the corresponding frame number and this frame number will be populated here.

So, whatever be either the page frame number comes from TLB or the frame number comes from this page table. So, this frame number is obtained and this displacement value comes and that finally, gives the physical address this frame number and this displacement that gives the physical address. That is how this page table, paging hardware is going to work with the TLB and all.

(Refer Slide Time: 14:39)

Now, this, so this speeds up the process definitely, so you can calculate what is the effective access time. So, associative memory lookup, so suppose say it takes epsilon amount of time. So, normally it is less than 10 percent of memory access time, so associative memory access is pretty fast.

So, it is in most of the cases it is less than 10 percent of memory access time and this hit ratio is say equal to alpha, that is the percentage of time that a page number is found in the associative registers. So, that is called a hit ratio and ratio related to number of associative registers.

So, the effective access time, so for alpha fraction of accesses, so it is found in the associative memory, so 1 plus epsilon into alpha. So, that is that is the time needed for accessing this for accessing this cache, and this then this. So, if it is not found if it is not found then that is the that hit ratio is 1 minus alpha. So, that 1 minus alpha, so this 1 is basically for memory access.

So, if the memory access time is taken to be 1 time unit, this is taken to be one time unit. So, the time when you have found a match in the cache itself then, so this the time needed is once from the cache it gets a hit. So, that takes epsilon amount of time and after that you have got to access the corresponding physical memory ok.

So, after getting that physical address calculated, so you have to access the memory, so that way it is 1 plus epsilon. So, this epsilon is for the cache access and this 1 is for the physical memory access then, so, so this is multiplied by alpha because alpha is the hit ratio.

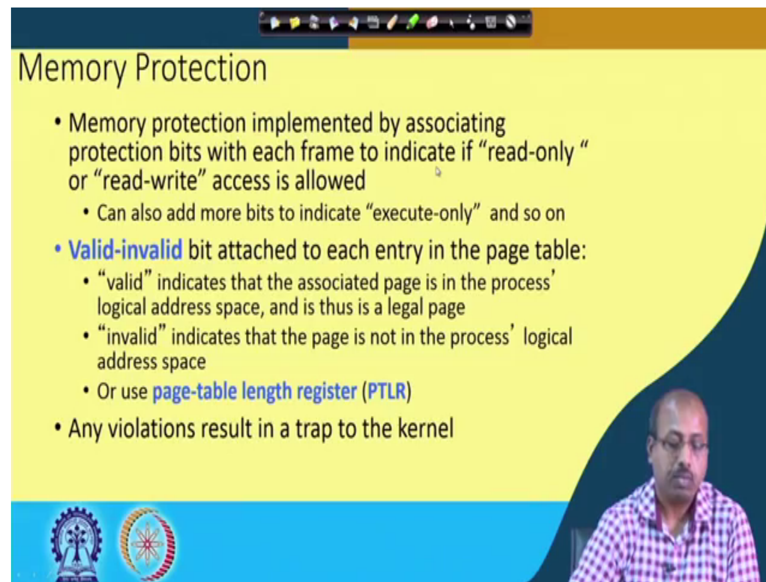
Now, other possibility is that the item is not found in the cache, so that probability is 1 minus alpha. And in that time, so you have spent epsilon time searching for the searching in the cache, then you have spent a time 1 time unit for this page table search, 1 time unit for this page table search, and another time unit for the physical memory access actual memory access.

So, this is, so this is going to be 2 plus epsilon into 1 minus alpha, that makes it 2 plus epsilon into 1 minus alpha. In fact, this epsilon in that case, you can say it is going parallel with this 1 of this page table access, so this epsilon can be ignored also for some simpler calculation. So, if I have got epsilon to be 20 nano second for this TLB search and 100 nano second for memory access, now alpha equal to 80 percent, so this is 0.8 into 1 plus epsilon.

So, 1 plus epsilon will make it 100 nano second for memory access and, so this overall formula becomes 2 plus epsilon minus alpha. So, if you put into this expression, so this is this is this is about 120 nano second. On the other hand if we consider this more realistic hit ratio which is about say 99 percent, in that case this expression will turn out to be about 101 nano second.

So, as the hit ratio is improving, so we have got much faster access to the memory ok. So, this way we can have effective access time calculated for the this TLB based access and see what is going to happen in the memory access.

(Refer Slide Time: 18:52)



This then this paging based mechanism, so it also provides procedure for giving memory protection. So, this is implemented by associating protection bits with each frame to indicate if read only or read write access is allowed. So, some of the pages we want that it should be read only.

So, nobody will be able to modify the page, and some of the pages it will be read write page where the pages may be modified. Normally, for the code part, so if a process has got code pages. So, code pages are not modifiable, so they are read only and that data pages, so they are read write; so, we have got this code pages and data pages read only and read write.

You can also add more bits to indicate execute only, so it is some protection we can provide. For with each page, so we can provide these additional bits to indicate the operations that can be done on the page. Then the valid invalid bit attached to every entry in the page table, so valid indicates that the associated page is in the processes logical address space and is a legal page, and invalid indicates that the page is not in the processes logical address space, so that is there.

So, if I if I keep a maximum size of this page table then these valid invalid bits may be useful, or we can use some page table length register that we have discussed previously. So, that checks given the page table it checks whether it is less than the PTLR value, so any violation it will cause a trap to the o s.

(Refer Slide Time: 20:27)

That way we can do this memory protection, so it is like this. Suppose, a process has got only 6 pages page 0 1 2 3 4 5, but in my system, so I have for every process, so I am giving it 8 page table entries ok, so page table, so size is kept as 8. Then what will happen, so we have got this corresponding to page 0, so the corresponding frame number is there, page 1 frame number is there and we have got a valid invalid bit.

So, up to page number 5 we have got this bit set to valid and for 6 and 7 they are set to invalid, because this 6 and 7 pages for this particular process it does not have those pages, so it is meaningless. So, we do not have their corresponding frame numbers and their corresponding the valid invalid bit is set to be invalid.

So, this is one type of protection mechanism that we can have that can be very easily implemented using this page table. We can also have protection like read write read only etcetera, so we can have another bit here ok, so we can we can put another bit here another bit here.

So, like that, so there you can tell whether it is a read write or read only like that. So, may be if I say if it is a read write, then the value is 1; if it is a read only then the value it is value is 0 may be this pages are read only and this pages are read write, so you can do it like this. And these are invalid, so it does not matter.

So, we can put some additional bit, so what type of whatever type of protection you can think about. So, you can you can put the corresponding type of bits into this page table entry and accordingly you can get the scheme implemented in terms of this memory page protection and all.

(Refer Slide Time: 22:21)

You can also share the pages, so one copy of this read only code shared among processes like this text editors, compilers, window systems. So, basically if I, if there are 10 people who are doing this editing job, so instead of having 10 different editor code copy, so we can have only one copy of the page and then that that may be shared by a number of users.

Now, if a page is shared, so then we must have the permission for all those users. So, if I set the page type to be shared then we can from the page table we can, so each processes page table it can have an entry for the corresponding frame and it can have this sharing information the read only. Similar, to this multi threads sharing of the same process space, so here also we can have that thing.

We can also use this for inter process communication if sharing of read write pages is allowed. So, we can do some inter process communication by this, so maybe we can have this one particular page is, so maybe this is a page which is shared by 2 processes P 1 and P 2. So, the page table for P 1 will have an entry which points to this similarly the page table for P 2 it will also have an entry which points to this.

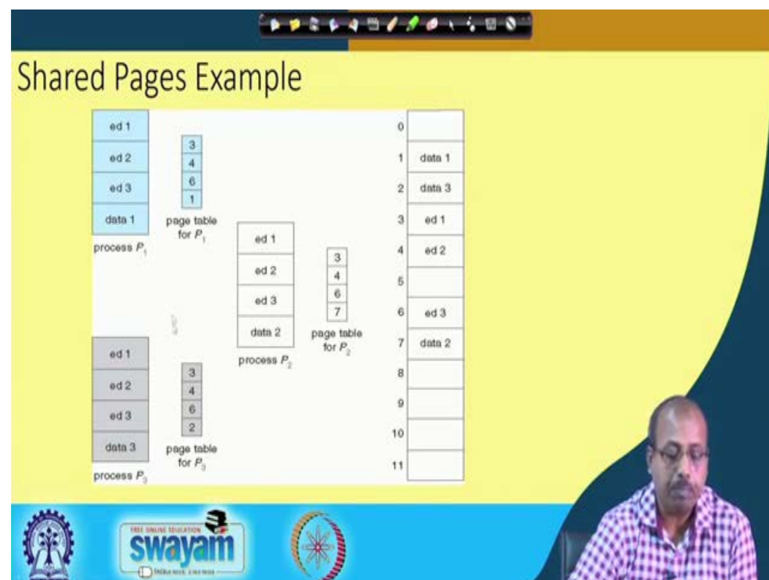
Now, since this is shared, so this P 1 and P 2, so they can access this particular page as a result implementation of sharing becomes simple, so that way we can have this thing. Similarly, inter process communication, so if you want to send some value from P 1to P 2, so it can be done via this shared page. And as I was telling this for the read only, so if this page is a is a copy of that editor software. So, this can be set to be of type read only, and then both of them both the processes can share that page.

Then we can also have private code and data. So, each process keeps a separate copy of the code and data, the pages for the private code and data can appear anywhere in the logical address space. So, it may so happen that when I am doing the editing job, so this is the editor code is same but the edited the editing or jobs that are being done, so they are different.

So, may be the process one it is modifying, so this buffer space for this editing job, so it is pointing here for the modification. Similarly, this P 2 may be doing the editing job here. So, they are sharing this code, but these 2 pages, so they are separate.

So, this concepts can be very easily implemented, so if you have got this paging mechanism available then this concepts can be very easily implemented. So, that way we can have this page sharing also both for sharing that code and also for ensuring privacy of the code and data, so we can do all this things.

(Refer Slide Time: 25:31)



So, next we will be looking into an example, like say these are the editor as I was telling. So, these are the processes P 1 and say processes P 1 and P 2 and P 3, so they are all doing this editing job ok. So, process P 1 has got this ed 1, ed 2 and ed 3 pages which are which are the physically located at frame numbers 3 4 and, so 3 4 and 6, so they have got this editing code.

Now this data which the editing job that process P 1 is doing, so that is at page number one and this page number 1. And this page number 1, this page number this page number 4 sorry page number 4 is actually the frame number 1.

So, this frame number 1 has got the corresponding data here. Similarly, process P 2, so this is also doing editing, so this ed 1, ed 2, ed 3, so these pages are same as the pages of this process P 1. So, the corresponding page table has got entries 3 4 and 6, but that data part is different, so data part is having 7. So, this is the corresponding memory frame number is 7, so it is doing the modification here.

And this one process P 3, so this is doing ed 1, ed 2, ed 3, so this is also doing editing job and this pages are shared 3 4 6 are shared, but data 3 is a different. So, that is the data part data segment for this particular process. So, this way we can have this sharing of pages and at the same time we can have this private data and code there.

(Refer Slide Time: 27:10)

Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space
  - Page size of 1 KB (2^{10})
  - Page table would have 4 million entries ($2^{32} / 2^{10}$)
  - If each entry is 4 bytes -> Page table is of size 16 MB
    - That amount of memory used to cost a lot.
    - Do not want to allocate that contiguously in main memory
- What about a 64-bit logical address space?

So, now if next we will be looking into the structure of the page table, so memory structures for paging can get huge using straight forward methods. So, this is a this is a problem with the paging, like we said that if this page size is small and the number of the logical address space is large. So, what suppose the CPU the address the logical address generated is say 32 bit and then the page size is 1 kilobyte.

So, then; that means, page the number of bits used in the offset part is only equal to 10, so this is your d part and this is the P h part. So, this d part, so since it is 1 kilobyte, so this is this is containing only 10 bits, but the address space is 32 bit, so this page is 22 bit.

So, the page table would have 4 million entries, so 2 power 32 divided by 2 power 10, that is 2 power 22, so that is about 4 million entries. So, that is a huge number of entries that will be there, and each entry if it requires 4 bytes of information 4 bytes of space then the page table itself will be of size 16 megabyte. So, that is a huge amount of memory that will be required and that amount of memory used to cost a lot definitely and it is very much likely that the process is they will not use that much of memory.

So, how to solve this problem? So, we do not want to allocate that contiguously in main memory. So, that definitely, so this for the page table ok, so even if some process is utilizing large amount of space, so just to hold it is corresponding page table. So, we cannot afford to hold 16 MB there this 16 MB of space cannot be given to give allow store the page table. And that is about 32 bit logical address, so if it is become 64 bit logical address then the number is definitely much larger than the previous one, so that gives problems.

So, this issue can be resolved by going into some different organization of the page table one of them is known as the hierarchal page table we have to go for a hierarchal structure of the page table. And then there is another option called a hash based hashing scheme based approach for this page table organization.

So, will see this schemes in subsequent classes like when we have go into this or a page table organization in more detail. So, paging helps us, but at the same time we have to look into this overheads that are coming and somehow we have to address this issues for the paging scheme to be successful, so will continue with this in the next class.