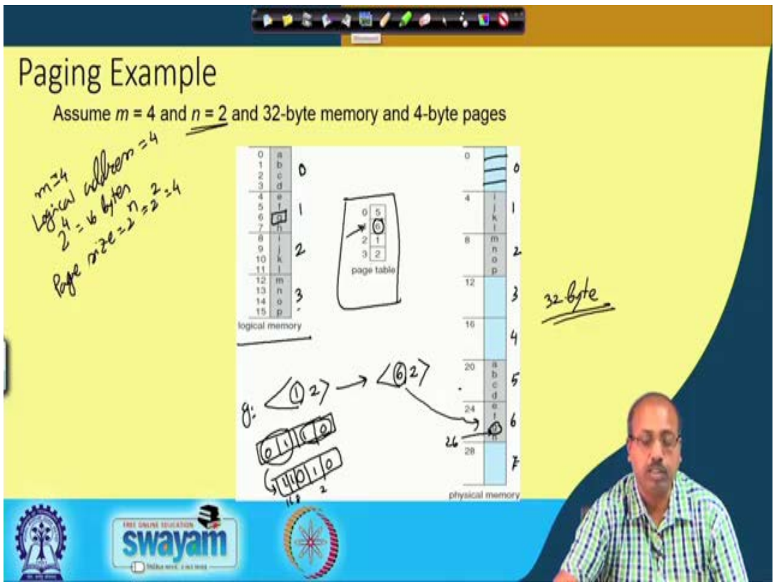
Operating System Fundamentals
Prof. Santanu Chattopadhyay
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture - 47
Memory Management (Contd.)

So, in our last class we were discussing about this paging scheme for Memory Management.

(Refer Slide Time: 00:32)



Paging Example

Assume $m = 4$ and $n = 2$ and 32-byte memory and 4-byte pages

So, today we will first look into that paging example once more and to make it more clear and then we will go to this page table organization and all. So in this particular example we have got we assume that this m which is the address generated by the CPU the logical address. So, logical address is 4-bit. So, logical address that is 4 bit. So, the if a maximum size of a process is going to be 2 power 4 equal to 16 bytes, assuming that this memory is byte organized. So, this is 16 bytes. So, this is the logical address space of a process.

So, and we assume that n equal to 2 that is page size is equal to 2 power 2 page size is 2 power n that is a 2 power 2 equal to 4. So, page size is equal to 4 bytes. So, we have got this byte 0, 1, 2, 3 that constitutes the first page then 4, 5 or say page 0 then this 4 bytes 4, 5, 6, 7 that constitutes page 1, this is page 2 and this is page 3. So, these are the four pages of a process that we have in the logical address space.

Now, the main memory so, that is our physical memory, so, that is considered to be a 32-byte memory. So, that actually shows that it is not mandatory that this logical memory should be equal to physical memory or logical memory be so, it is not necessary that way. So, this that is 32-byte, but what is required is that this frames of this physical memory they should be of same size as the pages that we have in the logical address space. So, this frame so, these are actually the frame. So, this is frame 0, this is frame 1, this is frame 2, 3, 4, 5, 6, 7 like that. So, these are the frames of the physical memory or main memory.

Now, frame size is same as page size. So, this can hold four variables similarly this can hold four variables. So, each of them can hold four variables. Now, we have got a page table that gives us the mapping that given a page number what is the corresponding frame number. Now, when this program was loaded into main memory so, it might have happened that these are the frames which were free and accordingly this frame this page 0 was loaded into frame 5.

So, in the page table at the index 0, we stored the value 5 which is the corresponding frame number. Similarly, at index 1 we store the page number we store the corresponding frame number 6. So, this page 1 was loaded at frame 6. So, this is 6, then at a page 2 we this page 2 has been loaded in frame number 1. So, this is 1 and this 3. So, that is page 3 has been loaded into frame number 2. So, this is the thing.

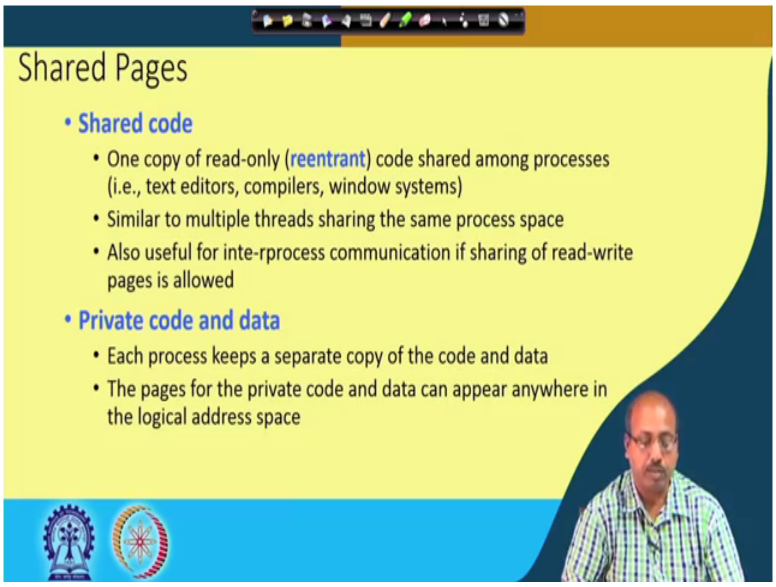
So, this way you see that now any address that is generated by the CPU, suppose it generates the address to access this g so, g is generated as the page number 1 and the offset is 0, 1, 2. So, this is the logical address for g. Now, what happens is that it consults the page table. So, it looks into the corresponding entry this entry 1, that is 6. So, this logical address is converted into physical address which is 6 comma 2 fine.

Now, 6; so, this 6 is basically frame number 6. So, it comes here and then 2 so, 0, 1, 2. So, it comes to this g or in terms of this address bits. So, you see that g is a this is a 4-bit value. So, this is address 6. So, it is 0, 1 1 0. So, this is the logical address that is generated.

Now, this 0 1 part so, this is used to index into this page table and it comes up with the value 6 ok. So, this say if it is 6, so, that is 1 1 0. So, this part is converted into this physical address by this 1 1 0 and these two bits are copied from here. So, this translation from the 0 1 to 1 1 0 is coming from the page table and this 1 0 these two bits are directly copied here. So, we are getting the address 1 1 0 1 0. So, it is 1, 2. So, this is a 2 plus 8 plus 16. So, 16 plus 8 plus 2, so, that is 26. So, this if this location address you see this is 24, 25, 26 only. So, this address is 26. So, this way the mapping is getting done.

So, this is how the paging scheme works. So, next will be going to our discussion on the page table that we were doing previously and as we have seen that page table helps us in doing sharing.

(Refer Slide Time: 05:57)

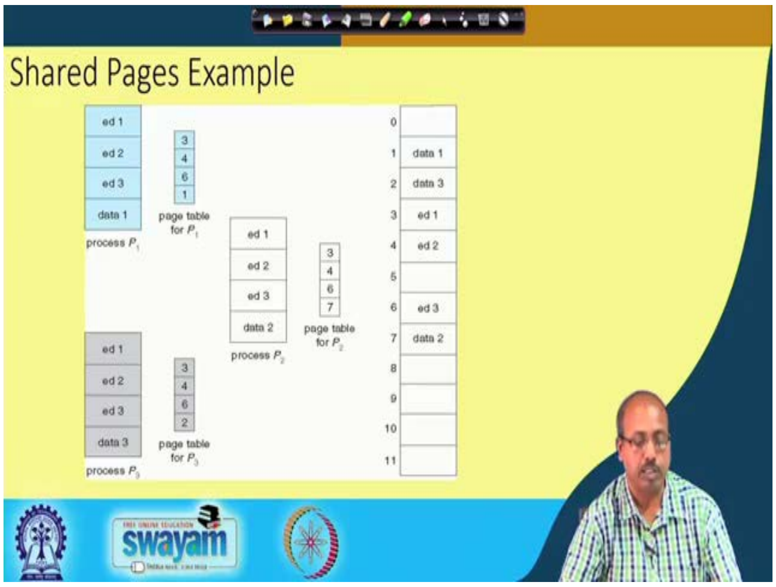


Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for inte-rprocess communication if sharing of read-write pages is allowed
- Private code and data
  - 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, you can shares the code among a number of processes by controlling the page table entries and we can also control the private code and data of individual processes by means of again by the page table.

(Refer Slide Time: 06:14)



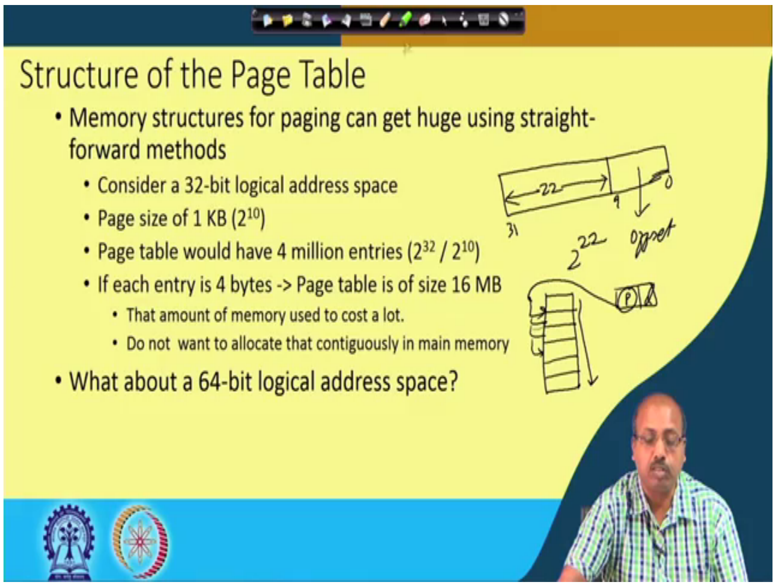
Shared Pages Example

So, by page table entries can do many things. Like in this example we have seen that we have got three processes: process 1, 2 and 3 and all of them are doing editing job. So, this ed 1, ed 2, ed 3 so, they are actually the code pages. So, this code pages have been loaded for into this memory frame 3, 4 and 6. So, this 3, 4, 6 so, these two frame number. So, these three frame numbers are common in all the three page tables, but this data part so, data part they are shared like they are exclusive.

So, basically this data part is specific for a particular process. So, data 1 for process P 1. So, this is having the corresponding frame number 1. Similarly, data 3 has the frame number 2 and data 2 has got the frame number 7. So, accordingly those page table entries have been created.

And, now, this code so, we are loading the editor pages only once for the entire system. So, it is not doing it exclusively for individual pages, individual processes. So, that way we can reduce the overall size of the of memory requirement by sharing the pages across the processes.

(Refer Slide Time: 07:33)



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?

Now, structure of the page table. So memory structures for paging can get huge using straight forward method. So sometimes we can see that the page table size is growing significantly. So for example, suppose we have got this 32-bit logical address space which is quite common. In fact, in today's processers so you will find that the address bus is 64-bit or 128-bit like that. So 32-bit address bus is pretty old I should say.

So, with that so, if we consider the page size is about 1 kilobyte, so, that is a 10 bits will be used in the address in the logical address. So, if the whole logical address is 32-bit, out of that this so, it is 0 to 31. Out of that this 0 to 9 so, this bits are used for the this for page offset. So, this is the offset part and if we are thinking about a straight forward way of this paging scheme then the remaining 22 bits so, they are devoted for the page number.

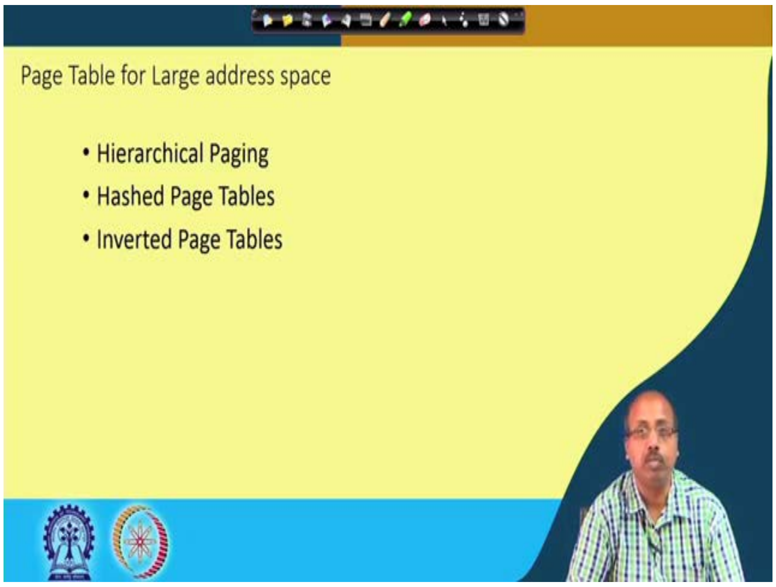
So, with that we can have 2 power 22 number of pages ok. So number of pages will be pretty high and page table will have 2 power 22 entries. So, if a process can literally be so big so, it will have 2 power 22 entries that are possible. Now, each entry if I say that each entry is 4 byte wide because the each entry will tell me the corresponding frame number. So if I assume that for storing the frame number I need a 4 bytes then the page table will be of size 16 megabytes. So, that is the huge amount of space. So to amount of memory used to cost a lot so definitely.

And we want to so, and we do not want to allocate that contiguously in the main memory. So, far what we have said is that if this is the page table then we have implicitly assume that the page table has got contiguous locations, fine. So, and we are directly using the page number ok. So, this page number P and offset d, so, this page number part is used directly to index into this table.

Now, that is only possible when this the entire page table is allocated sequentially. So they are given sequential locations in the memory. So, they are contiguous locations in the main memory. But, definitely getting say 16 megabyte of contiguous space in the memory is a challenge ok. So, we cannot afford that.

And naturally the situation will be even worse. So, if we consider 64-bit logical address space because that is going to be even much more than what we have so, possibly in the memory. So, the and also many times we do not need such a big page table such a big page table may not be present. So, we have to do something so that for larger the processes we can still allocate this page table space and possibly in a non contiguous manner. So, that is what we will try to see.

(Refer Slide Time: 10:52)

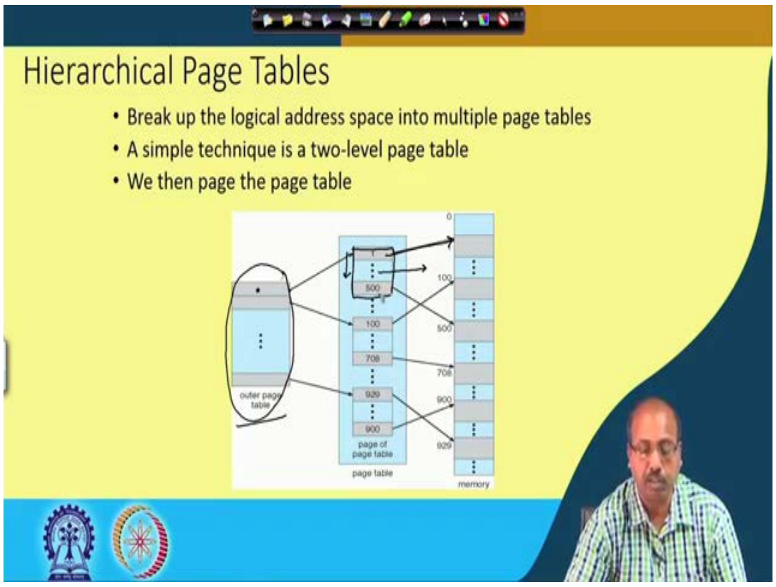


Page Table for Large address space

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

So, how can we do that? So, for that purpose so, there are three different techniques that have been proposed by this page table designers. So, for large address space so, one is called hierarchical paging, one is called hashed page table and another is known as inverted page tables. So, we will look into them one by one.

(Refer Slide Time: 11:12)



Hierarchical Page Tables

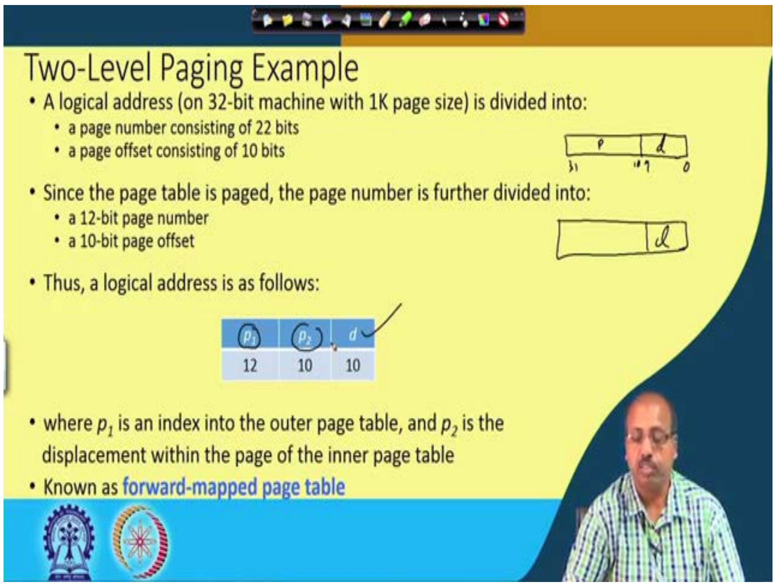
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

So, this is the hierarchical page table. So hierarchical page table says that we break up the logical address space into multiple page tables. So and then a simple technique is a two-level page table like here you see that we have got; we have got this outer page table. So, an index so an entry here so it gives me; it gives me a pointer to another page table.

So, previously what was happening is that this entry was directly pointing to a particular page in the main memory, but here that is not the case. So, this in turn points to another page table and I have to index into this page table further to come to a particular entry. So, if this is; so, this actually gives me the page number. Similarly, each of them it will give me some page number.

Now, you see that the I need contiguous space only for this outer page table. Now, this in inner page tables so, each page table can be located separately. So, that way so, I do not need say that the 32-bit address space. So, I do not need 2 power 22 or 16 megabyte of contiguous space. But, if I assume that my page table individual page tables are 500 entries; so, 500 into 4 so, 2000 byte contiguous location, that will be sufficient ok. So, that way I can allocate them on different page tables. So, this is the way we have got a hierarchical structure.

(Refer Slide Time: 12:47)



Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

p_1	p_2	d
12	10	10

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as forward-mapped page table

So, to explain it further so, we can have a look into this two-level paging example. Suppose going back to a previous example that we have got a logical address space with 32-bit machine with 1 kilo page size. So, page size is 1K and we have got this logical address space is 32-bit.

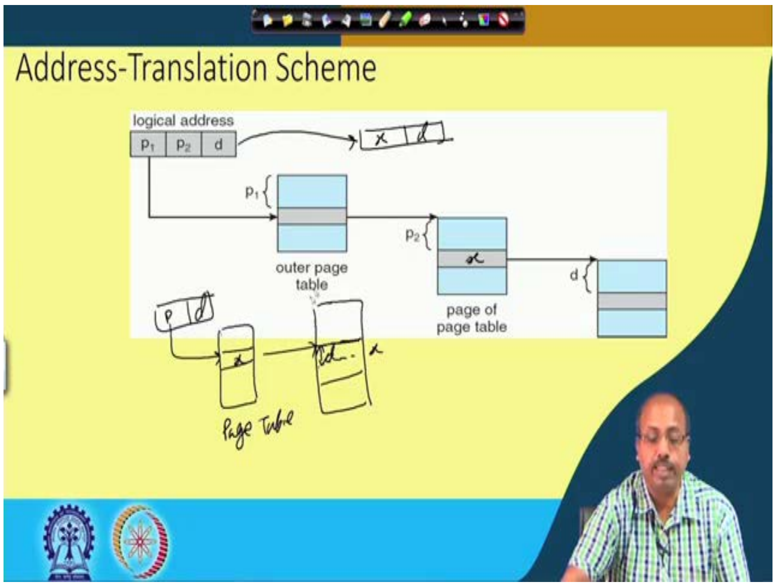
So, this is divided into the page will consist of 10 2 power 10 entries or 1K entries. So, that the page offset will have 10 bits and this and this page number part so, that will that will take 32 22 bits. So, as we were discussing previously so, that 32-bit so it is divided into two part; one is the 10 bit of offset 0 to 9. So, this is the offset part d and 10 to 31; so, this is the page number part p and since the page table is also paged, then the page number is further divided into a 12 bit page number and 10 bit page offset.

So, it is further divided into two parts. So, the structure now becomes something like this. So, we have got this displacement part d here, then we have got this p 1 and p 2. So, p 1 is an index in the outer page table and p 2 is an index is the displacement within the page of the inner page table. So, this inner page table so, if we assume that each page table entry has got 2 power 10 entries in it, page entries in it, then it is going to have a structure like this. So, that overall the 22 bit it is further divided into 12 bit for outer page table and 10 bits for inner page table.

So, this type of page table structure so, they are known as forward mapped page table. So, here actually now what is happening is that from one page table structure one page table we are it is pointing with the next level of page table and that can continue for quite some time quite some level. So, if it is even higher say 64 bit then you can break it down further so that we can have more hierarchical representation. So, that is.

So, this way we actually solve the problem of allocating contiguous space for the page table. So, the page table can also be divided into pages and they can be distributed over different regions of the memory.

(Refer Slide Time: 15:14)



Address-Translation Scheme

So, this is the thing. So, p 1 is actually indexing into the outer page table and then outer page table from the outer page table you get the address of the inner page table. So, then the inner page table the number that is there.

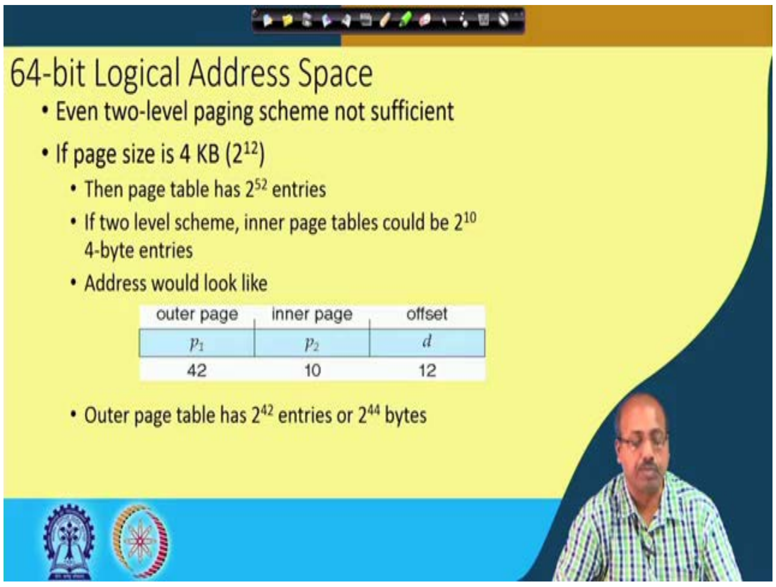
So, from there we come to this address and then this p 2 is use to used as an offset with in this inner page table to come to the actual page table entry so, for this particular address and that page table contains the corresponding frame number. So, this particular entry it contains the corresponding frame number and then that frame number and d so, that gives me the physical address.

So, ultimately what is happening is that if you have got this entry as say x. So, this is converted into a physical address where this is a one part is d other part is x, but this conversion is done in two phases. Previously what was happening is that from the we have got this p and d. So, this p was directly indexing into this page table. So, there was a single level of page table.

So, it was indexing into that and that was giving me the x actually. So, it was now going to main memory and from this so, it is the; it is the beginning of this frame x. And within that and offset of d so, this is the offset d that we have. So, this x d so, that was giving me the actual address from which we have to access it.

So, this mechanism the problem that we faced is this page table size was becoming pretty large. So, we have divided into a number of levels and in the hierarchical fashion. So, we can solve this problem of large page table.

(Refer Slide Time: 17:08)



64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like

outer page	inner page	offset
p_1	p_2	d
42	10	12

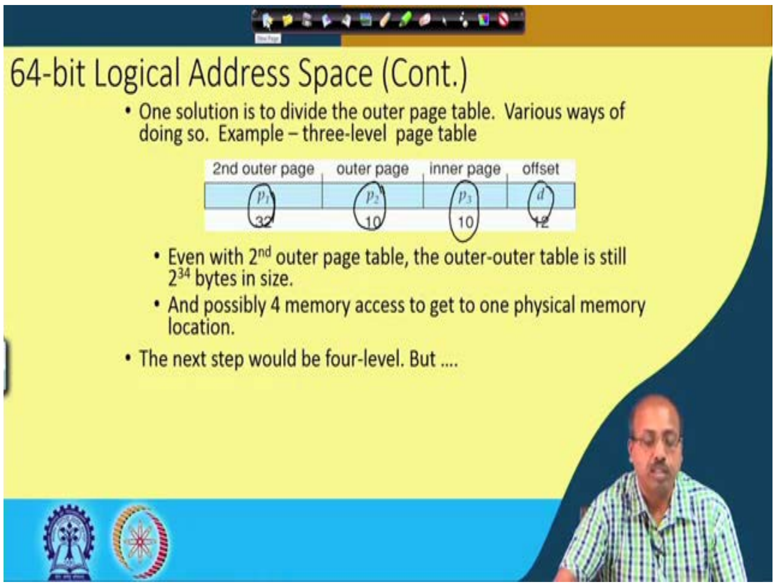
  - Outer page table has 2^{42} entries or 2^{44} bytes

So, for 64-bit logical address space, so, two-level scheme is not sufficient like even if we assume that page size is say 4 kilo byte then this page table offset part will have 2 power 12 it will have 12 bits offset part it can access one of the 2 power 12 entries. Then the page table has 2 power 52 entries. So, because total logical address is 64-bit, out of the 12 bit is taken as offset within the page. So, remaining 52 bits so, they are going to identify the page number. So, page table has got 2 power 52 entries in it.

Now, even if we are having a two level structure then the inner page table could be 2 power 10 4 kilobyte 4 byte entries because this if we assume that each entry in the page table takes 4 byte of space because that is the address. So, if we say like that then it can accommodate only 2 power 10 entry. So, 2 power 10 that inner page table can have or 10 bits, then the outer page table will have again 42 bits to identify a particular inner page table ok. So, that way the outer page table size is 2 power 42. So, that is a; that is a very important problem.

So, that has to be solved. So, this is a 2 power 42 entries and each entry is 4 byte. So, total 2 power 44 bytes will be needed just to store the outer page table. So, that way this two-level scheme is not sufficient for this 64-bit addressing. So, we need to go for even more number of levels for this type of structure.

(Refer Slide Time: 18:48)



64-bit Logical Address Space (Cont.)

- One solution is to divide the outer page table. Various ways of doing so. Example – three-level page table

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

  - Even with 2nd outer page table, the outer-outer table is still 2^{34} bytes in size.
  - And possibly 4 memory access to get to one physical memory location.
- The next step would be four-level. But ....

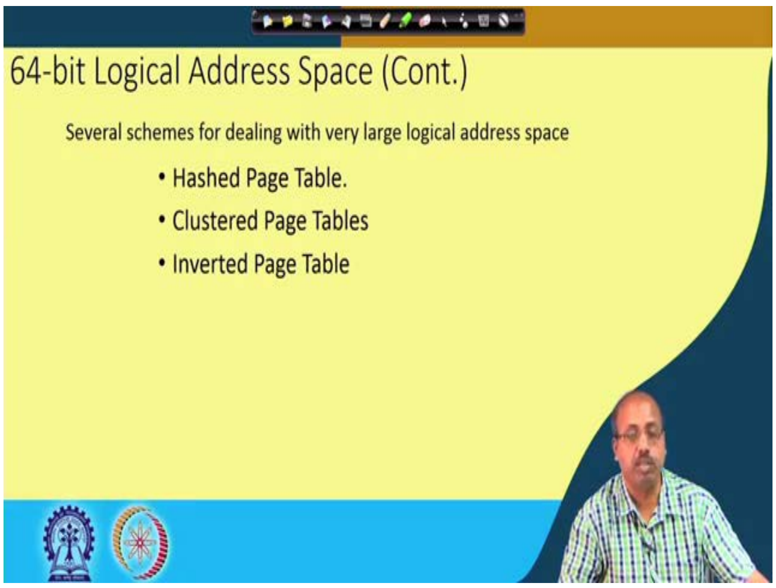
So, one solution is to divide the outer page table. Now, we have got a second outer page table. So, we have got this offset then inner page, outer page and then we have got the second outer page. So, we have got a 3 level page table. So, even with this is three-level page table the still the second outer page table still contains 2 power 34 bytes because there are 2 power 32 entries. And each entry is 4 byte if we assume like that, then there are 2 power 34 bytes needed for the outer page table the second outer page table.

So, possibly 4 memory access to get to one physical memory access. Now, what is happening is that for a particular address so, it has to first look into this outer page, then it has to second outer page, then from there it will come to this, then from there it will come to this, then from there it will come to this and then finally, it will come to the actual memory location.

So, this after going through this p 1, p 2, p 3 – three memory access for three levels of pages it will only come to the actual frame number of the memory. And then from the frame number frame number and offset so, that will give me the total the actual address and then that will the that location will be accessed. So, it requires a 4 number of memory accesses for one physical memory location. So, memory access time is becoming four-fold the scheme in which there is no paging. So, that is very costly.

So, this way it can go on, but the for next level will be four-level. So, even if the if this is considered to be a high amount of contiguous space then we can go for four-level space, but going like that again number of hops or number of steps in accessing a particular memory location so, that is increasing further. So, that way it will go on increasing to this logical this number of levels for memory access. So, it is going to increase further. So, we have to see how is this problem can be solved. So, this is still not the good solution.

(Refer Slide Time: 20:51)



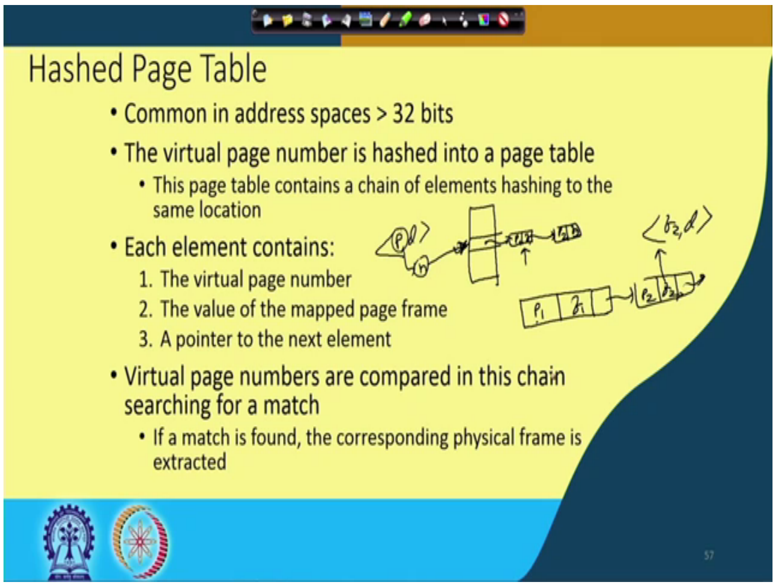
64-bit Logical Address Space (Cont.)

Several schemes for dealing with very large logical address space

- Hashed Page Table.
- Clustered Page Tables
- Inverted Page Table

So, coming with some other possible techniques for which this large physical address space can be resolved; one is known as hashed page table, then there are clustered page tables and there are inverted page tables. So, we will look into them one by one for 64-bit logical address space so, we have to follow this schemes.

(Refer Slide Time: 21:13)

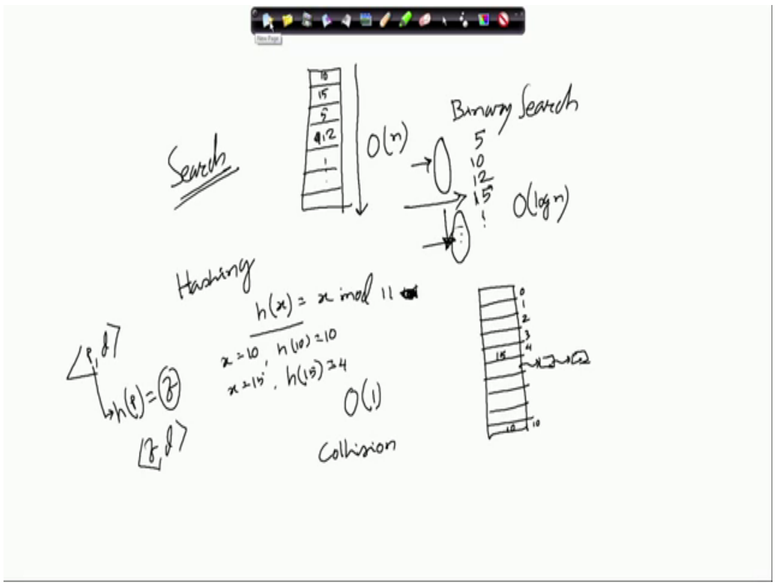


Hashed Page Table

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains:
  1. The virtual page number
  2. The value of the mapped page frame
  3. A pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

One is the hashed page table. So, common in address space is greater than 32 bits because then we have got large number of pages and the virtual number of virtual page number is hashed into a page table. So, I will just quickly give you an idea about hashing. So, it is like this.

(Refer Slide Time: 21:33)



I have got an array and in this array I have stored some numbers. I have stored some numbers. Now, this is say 10, 15, 5, 12. So, like that there are numbers. Now, I have to search into this array. Now, if we want to search it, so, one possible way is that you start looking from one end then go to the other end. So, that way the complexity will be of the order of n fine.

Now, other possibility is to do a binary search. So, in a binary search what we do? We have got this array sorted this numbers are sorted. So, 5, 10, 12, 15 so, like that and at the beginning we probe at the middle of the array, we check at the middle of the array. The number that we are searching if it is less than the number at the middle, then we restrict our search in this region.

So, again you probe at the middle of this upper part and try to see the whether it is matching or not. And if the number is greater than the number at the middle or the probe position then we loop into this lower part and try to see if there is a. So, try to see at the again at the middle whether there is a match or not. So, in this way if there is a match then in time log of n so, you will be able to find it. But, even this one the problem is that you have to keep that array sorted and then you have to spend log of n time to search for the number.

Now, another way of doing it is known as hashing. So, in hashing what we do we use a function a computational function which depending upon the data value that we give so, it computes and address of it. For example, one possible hashed function may be like this. So, this is x modulo say 11 plus 1. So, or may be x modulo 11. Now, any set of numbers that are given to me so, number 10 if I if x equal to 10 then h of 10 is equal to 10. So, if I give x equal to 15, then h of 15 is equal to 4 by passing through this function x mod 11 so, that is equal to 4.

Now, so, while I am storing this numbers into this array so, wherever it whatever this address is computed so there we put this entry. So, this is say the index is a 0, 1, 2, 3, 4 like that. So, at this entry we put the value 15. Similarly, at the 10th entry I will put the value 10.

Now, if I am looking for a number then I just pass that number through this function h and then that gives me an index and I check the corresponding index value. So, if the number of there in the table then I will be able to find it immediately. So, in a order 1 time I will be able to reach that particular number if the number is present.

Of course, there are problems like a set of numbers they can give the same value computed by this h function or the hashing function. So, that is known as the problem of collision and if a collision occurs then there are several mechanisms by which we can solve it. One possibility is that if so, you just put all those numbers in a chain at this in this at a all these collided number so, you put them on a list put them on a chain. So, that is one possibility.

Other possibility is that you store the number in the next available free slot on to this table. So, this type of collision resolution mechanisms are there, but assuming that the collision probability will be less and this even if collision occurs so, the number will not be very far from the location at which we have probed. So, that way it can give us quick access to the actual number.

So, the same mechanism can be followed here. Like we have got this page number and offsets. So, we have got this page number and offset; so, this particular page number. So, instead of going through this page; going through this page table and all getting the frame number so, we pass it through a hashed function; so, h of P. So, h of p so, this will give me the frame number f and then this f and d so, that will tell me what is the physical address of the memory to be accessed.

So, with this background so, we will be going in to discussion on this hashed function based page table organization or a hashed page table organization. So, the virtual page number is hashed into a page table and this page table contains a chain of elements hashing to the same location. As I was telling that the number of x values they can give me the same h of x values.

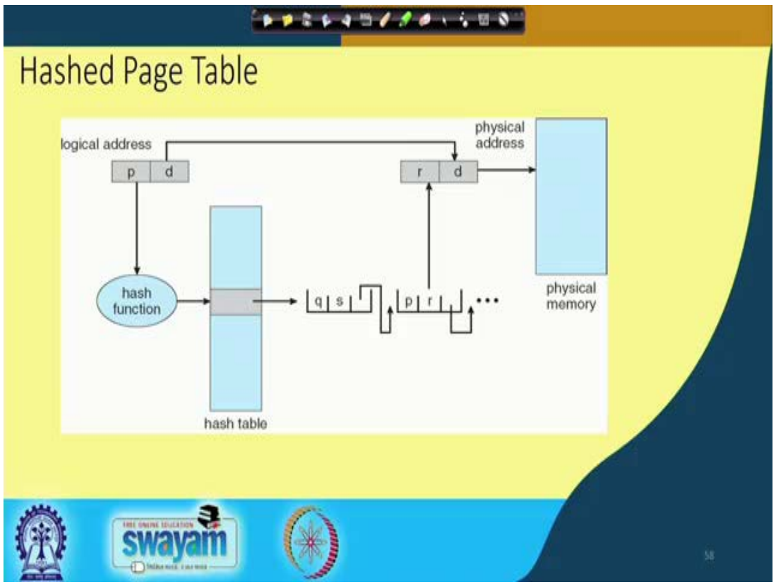
So, if I say that number of pages they give me the same page number then they are put on to the they are put on to a chain. So, the my page table structure is basically it is a chain so, it is a chain of all those frame numbers; all those page number, frame number combinations I should say the page number and the corresponding frame number P 1 f 1, then P 2 f 2 like that. So, with the idea that all these page numbers P 1, P 2 etcetera they all hashed to this particular index in the page table.

Now, this frame numbers can be found by traversing this list. Now, each element it will contain a virtual page number and then the value of the mapped page frame and the pointer to the next element. So, that is what I was telling. So, the pointer part I have not shown it explicitly. So, all these entries that I have, so, it is having this a structure. So, it has got the page number P 1 then the corresponding frame number f 1 and a pointer to the next entry. So, that way this list is organized. So, this is P 2 f 2 and then that point into the next one like that. So, this way it goes.

So, this virtual page numbers are compared in this chain searching for a match. Now, as I said that some address is generating by the CPU, P and d. So, this P is passed through the hashed function h. So, it is passed through the hashed function h and suppose it gives me this particular index of the page table, then we search through this list to see whether this P is equal to P 1 or P equal to P 2 like that. So, wherever I find a match that particular frame number is taken out. Suppose, I find that P is equal to P 2 then this is taken out and this f 2 and a d so, that gives me the actual address of the memory that we have to accessed.

So, virtual page numbers are compared in the chain searching for a match and if a match is found the corresponding physical frame will be extracted. So, that way it will be going to the corresponding physical frame. So, this is how this hashed page table works. The advantage that we get is that we do not have to have a say three – four levels of hierarchy for accessing a particular page. So, it is only a single hashed function calculation and then probably a chain and this the chain will not be very large because it is only the colliding pages that will be coming on to that chain.

(Refer Slide Time: 29:26)



Hashed Page Table

So, this is the structure. So, the logical address that is generated the page number part of it, it passes through this hashed function that gives me and index into this hashed table. And that hashed table index so, it has got a list of this memory frames where it they are matching. So, this is q. So, q is actually one virtual page.

The q and p, so, these are two virtual page number that the that have hashed to the same physical page physical memory page number frame number. So, and they are actually they have hashed to the same index of this page table not the frame number. They have hashed to the same index of this page table.

Now, it comes to this page q has been stored in memory frame s; page p has been stored in memory frame r like that. So, while searching through this list so, it finds a match for this p here and the corresponding frame number is taken out. So, this frame number and displacement so, that gives me the physical address. So, this way this physical memory will be accessed based on this. So, number of hierarchy that we have use only one level of hierarchy. So, that is how this is organized.

Of course, the challenge that we have is designing a good hashed function so that this so that this page numbers they map on to different different frame numbers for different different entries into this hashed table. So, that is the definitely a challenge.

So, we will continue with other the memory organization in page table organization in the next class.