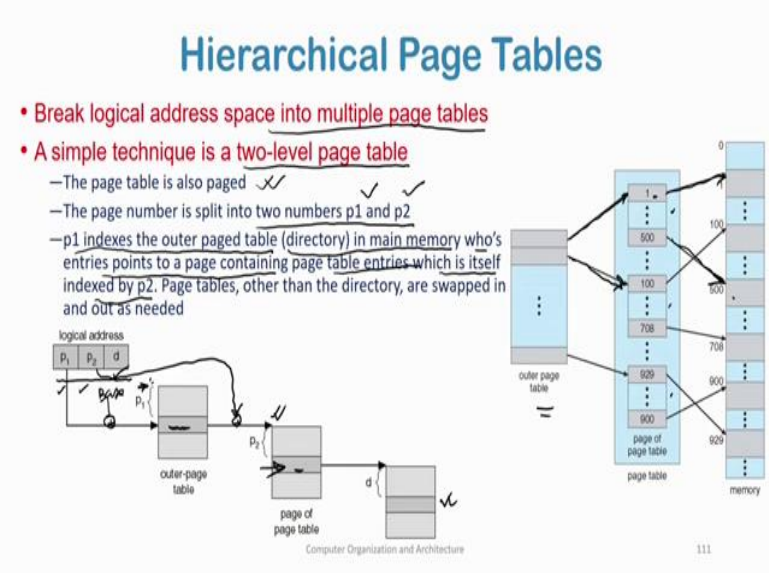
(Refer Slide Time: 26:03)



The next approach that is used to reduce page table sizes is by using hierarchical page tables. So firstly, what did we use? We used a page table length register. The which was without segmentation and then we said that typically the virtual address space has a stack part and a heap part to address and the page table length register only allows the page table to go grow in one direction.

So, we addressed that by having two segments; one containing possibly the stack, the other containing the heap and each of these two segments has two page tables and the so, by directions two possible directions of increase for the process becomes available.

So, then what happens? We came to the segmentation. So, instead of having just two segments I divided into multiple segments. So, therefore the even if the processes address space is very scattered, so the modules tend to be clustered in their address spaces.

So, I have separate page tables for each of these modules or segments and therefore, the overall size of the page table reduces. And then the segment table can be in the main memory and we understand that the page tables can also be page the page the page table can actually at a given time can be in the in the in the secondary storage as well.

So, within a given segment I go and I access a page table that page table may not be there in main memory at a given time. Then I bring that page table from corresponding to that segment from secondary storage to main memory and I then I access and then I access what the main

memory page frame. So, this also becomes available with the use of paging with segmentation. Then we come to a hierarchical page tables. So, hierarchical page tables the simple so the here we do not go into segmentation, we don't have segmentation here and but we have multiple page table levels; hierarchical page tables or multiple page tables multiple multi-level page tables. We have multilevel page tables.

So, we break the logical address space into multiple page tables. The simplest scheme in this is a two level page table. So, what happens in a two level page table? So, as I told you the page table is also paged similar to the paging with segmentation, the second and third level page tables may or may not be in main memory.

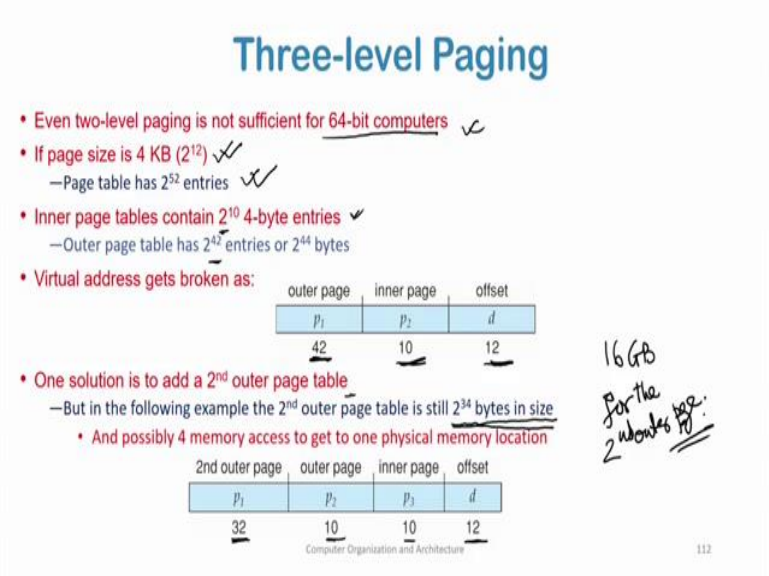
So, if this allows the page table to be paged and therefore better sharing of the main memory is possible. The page number is split into two; the page number is now split into two parts. One is P_1; the other is P_2; P_1 indexes the outer page table; P_1 indexes the outer page table or directory in main memory whose entries point to a page containing the page table entries which itself is indexed by P_2.

So, what happens? The logical address space is now divided into two parts; P_1, P_2 and this is the offset part. So, from P_1 I go to the outer page table ok. So, the page table the my page table base register tells me the position in memory of the start of the outer page table.

Now, based on that one adding that up adding this base with the P_1 that I have I get into where in the outer page table is the start in memory of the second page table P_2. So, start in main memory for the second page table P_2 is obtained in the entry for the outer page table. From here I from here I use this one from I now add this P_2; I add this P_2 with this value that I get and I get this one and I get this one where in P_2 contains my page frame number. When I get the page frame number I added to the data and I go into the main memory.

So, here is the outer page table. The outer page table points to a number of inner page tables ok. So, this is this one points to the page table 1, this one points to another page table ok; it points to a number of page tables and each inner page table then contains the frame number. So, this one tells me the this one tells me frame number 1, this one tells me frame number 500 and I go to the frame number and I added to the data and get the data required from the physical memory. I get the physical page frame number here, I access the page and I get the physical data.

(Refer Slide Time: 31:35)



Now, two-level paging is not always sufficient. So, even two-levels paging is not sufficient for 64 bit computers. So, what do we do? And why is that so? Let us say if a page is of size 4 KB as we had previously, then in 64 bit computers the page table will have 2^{52} entries ok. So, $2^{64} - 2^{12} = 2^{52}$.

So, therefore the page table will now have 2^{52} entries. The inner page tables contain 2^{10}. Let us say if the inner page table contains 2^{10}, 10 4 byte entries, then the outer page table has 2^{42} entries. So, if the inner page table contains 2^{10} entries.

So, my offset contained this 12 bits, so2^{12}, 12 bits. The inner page table contains what? 2^{10} entries; So, I use 10 bits. So, now, my outer page table has 2^{42} entries. So, this one is still 42 bits. So, it contains 2^{42} entries right.

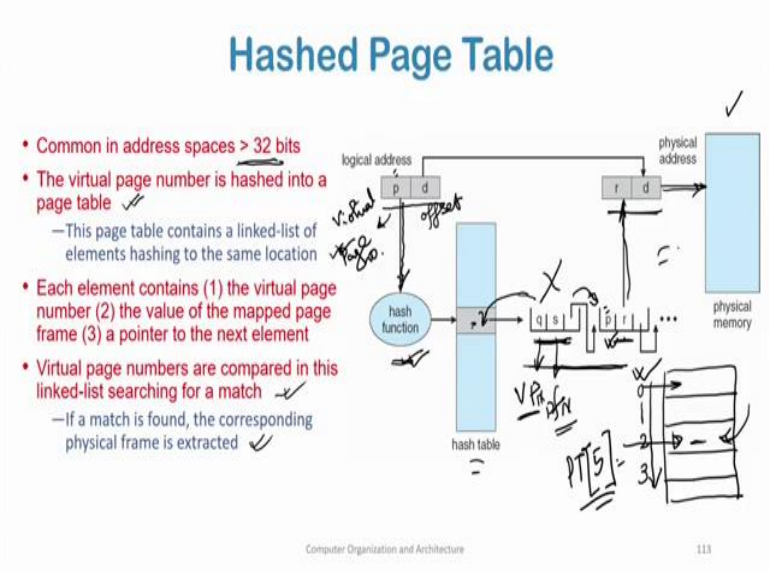
So, even if we have a two-level page table it will have 2 to the power this the outer page table will have 2^{42} entries which is very huge. Now, one solution is to add a second order outer page that is we use a 3 level page table ok. So, now what we have? We have a offset again 12 bits, we have a inner page table in inner page which is of 10 bits to indicate where in the inner page. Then again we have another outer page which is 10 bits and we also have a second order outer page which is still 32 bits.

So, but the following exam, but the following example the second order page table is still 2^{34} bytes in size. Why? Because I have 2^{32} entries and each such entry is 4 bytes or 2^2 bytes. So,

the size of the outer second order outer page table is 2^{34} bytes which is very huge. We understand that 2^{30} means 1 GB. So, if the therefore, 2^{34} will mean 16 GB.

So, 16 GB for the second outer page which is very huge, which is still very huge ok; Even with three-level page tables it becomes very huge. But oh obviously, this scheme is there because for larger um for larger address spaces; logical address spaces it shows that we may need to have multiple levels of page tables more than 2 ok.

(Refer Slide Time: 34:58)



In order to control in order to control size of another technique that is used to control the size of a page table is by using a hashed page table. It is commonly used for address spaces greater than 32 bits. So, this is more prevalent for 64 bit computers. The virtual page number is now hashed into a page table ok.

This page table contains a linked list of elements hashing to the same location. Now, this now what we have this logical address has this offset part and this is the virtual page number, virtual page number. Now, this virtual page number part of the virtual address or logical address is used as a hash function; this virtual page number is used as a hash function and it hashes into the page table.

Now, each entry in this page table contains a linked list of linked list of elements with which are obtained through by, so, this entry is what? By hashing this virtual page number by hashing this virtual page number I get to an entry and this entry contains a linked list of elements.

So, the virtual page number is hashed into a page table. So, this is the page table hashed page table or hash table and this page table contains what a linked list of elements hashing to the so, all these elements hash to the same location here; all these elements hash to the same location. And therefore, are linked together in a chain. It is a chain or a linked list as is done for hashing. So, from the hash function I get to a location, in this location I have a linked list of all elements that hash to this location.

Each element contains what the virtual page number, the value of the mapped page frame. So, this is the frame number, this is the page number virtual page number, this is the page frame number page frame number virtual page number and the third part is a pointer to the next element ok.

So, what do we have here? So, I have a virtual page so, what do I have? I have hashed into this one with the page number and then I match I match this virtual page number with the elements. So, I when I have come to this location I go to this linked list and one by one I find out whether this virtual page number matches with the virtual page number here.

So, for example in this case the first element does not match but the second element matches. This one is also p this one is also p ok. So, because this has matched I take this frame number which is r and I get the frame number r and I add it to the offset part which is d here.

So, this r plus d gives me the physical address and using the physical address I access the physical memory. Now, therefore using although this one is a per process; so, I have a hash table for each process here. However, however I need to keep only as many virtual pages as are needed by the program at a given time.

So, if I hash to this place and these elements will only contain those element will only be there which actually contains a physical page frame number here. So, if a particular virtual page if a particular virtual page is not there in physical memory for corresponding to a virtual page if it is not there in physical memory it will not be found here ok.

So, in this scheme we only keep those elements for which I have physical memory pages mapped to the virtual addresses. If I don’t have physical memory pages a map to the virtual addresses I need not keep in this hash table.

So, although this is a per process phenomenon; that means, this hash table is kept for each process, it is still smaller than the big page table structure that we had previously. Now, you may ask as to why even previously we had such a big page table and why not why is it not that I only keep those entries in the page table for which I have valid logical addresses.

This is because this is because this page table is indexed with the virtual page numbers. The page nibble is indexed with the virtual page numbers and let us say if I have to access a particular logical virtual page the searching this page table becomes very easy, when I have this sorted in terms of virtual page number.

This hashing makes it easy because hashing will actually get it through a through a through a small number through a through a small number of search through a small search I can actually get to the, get to the entry which will contain my page frame which get to the element which will contain my page frame due to the hashing that I have.

But if I did not have this hashing to search in the page table, where my particular page frame will reside. That is made easy by keeping an entry for each virtual page whether is it is in the logical address space of the of the process currently or not I keep an entry for all for all virtual pages that I have and then that makes it easy to search for a particular page table entry.

Because now, if I am search if I want to search for logical address numbers, a logical page number 5; virtual page number 5, I know that it is it can be indexed by page table 5. I can I can just go to page table 5 and I will get this entry ok.

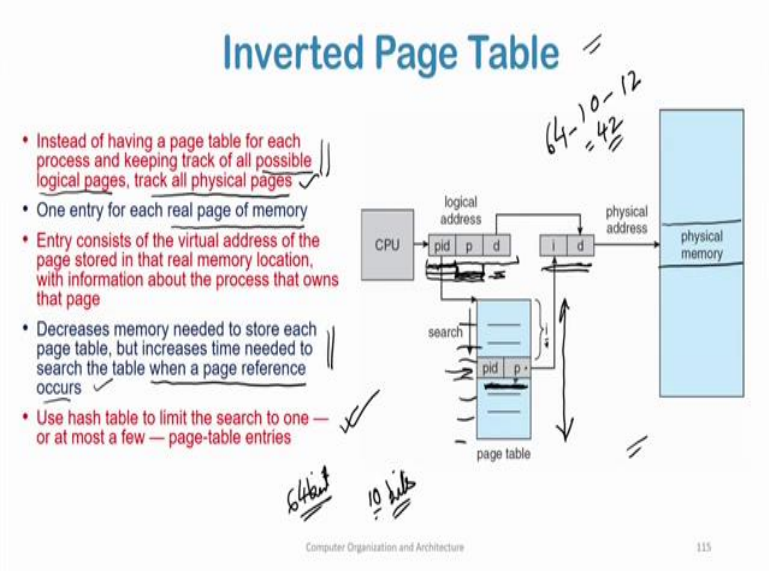
But, but then this will not be possible if I don't have a structure like this ok. So, if I have an added structure, now where will be the main problem? Let us say now I as I told you the logical address space of the process can grow with time during execution. Now I want to include a new page into the into the virtual address space of a process.

So, how will I insert this? This is not a problem if I have all entries in the in the page table. I just say that now this particular entry has become valid; the contents are now valid. But previously this page this virtual page which is indexed by 2 was not part of the virtual address space of the process.

But now, this page has been added to the virtual address space of the process and now this entry is valid; this will suffice for me. So, therefore, I have to keep the entire page table, if I don't

use a mechanism like hashing ok. Virtual page numbers are compared in this list searching for a match. If a match is found the corresponding physical frame is extracted as we discussed.

(Refer Slide Time: 43:34)



The next approach is the use of an inverted page table. Now, the in the main concept in inverted page table is the following: Instead of having a page table for each process and keeping track for all possible logical pages that we have, we only keep track of all physical pages. We only keep track of all physical pages. Now, I have one entry for each real physical memory and this page table is now indexed by page frame number because I have only one page table for the entire physical memory. It is not a per process page table anymore. So, I have one entry for each real page of memory.

So, therefore, this page table is now indexed is now indexed by page frame numbers. And what does it contain inside? It contains the virtual address along with the pid of a process. So, entry consists of the virtual address of the page in that real memory location. So, this particular physical memory location is pointed to by a particular virtual address of a particular process.

So, that virtual address and the process id both need to be kept in the entry of this page table. So, how now basically what has happened see, this index this index i is what? It is the page frame number; it is not the virtual page number this is the page frame number and this page frame number directly because of this index i is known, I can directly go to this i and add this offset and get the physical address. And in this location what do I have? I have the pid and the virtual address.

So, what is the advantage? It decreases memory needed to store page table, but increases time needed to search the page table when a page reference is made. So, we need to discuss as to how a page reference will be made.

The CPU still floats this logical address ok. Now this combination of pid and p. So, now the logical addresses, address again is divided into three parts because the address space is very. This is used for large address spaces for example, 64 bit 64 bit computers. So, basically I will keep let us say, let us say 8 bits or say 9 bits, 10 bits for the pid. So, I can accommodate at most 1024; if I keep 10 bits for the pid I can accommodate 1024 different processes; which is very large.

So, this combination of pid; that means, which process and what is the page virtual page number. So, if I keep 12 bits for this one; let us say 12 bits for my page offset and I keep let us say 10 bits for my process id. So, if I have a 64 bit system, 64 bit logical address I still have 64 - 10 - 12 which is = 42 bits for my virtual address which is very huge; So, no problem with this virtual address space that I have.

So, now what will I have? Given this pid and p; I will search this page table. I will search the entire page table in I will search the entire page table to find if I get a match for this p and pid. I will search from here, search the entire page table I so, I know the pid and the page number, virtual page number and the pid and I am going to search the entire page table to get a match for pid and p.

If I get this match, if I get this match I know wherein the page table I got this match which is i; where the index in to the page table where I got the match. And this index basically tells me the physical page frame number.

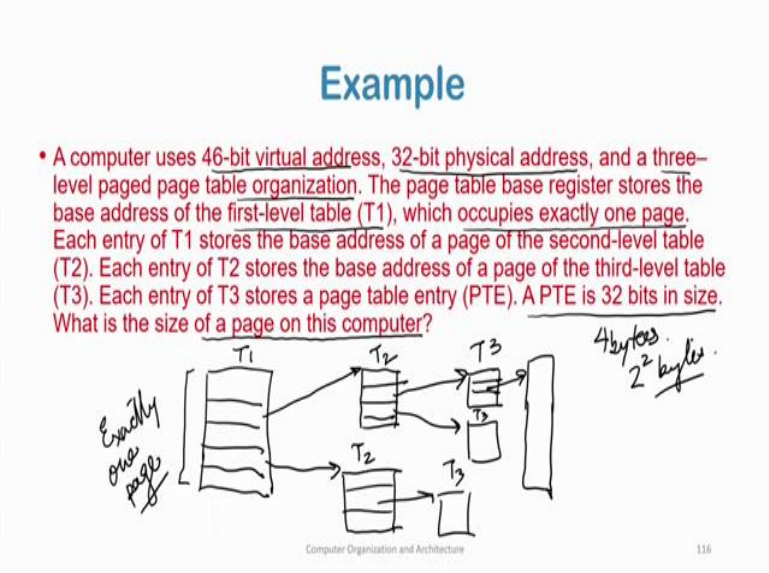
Now, I get the physical page frame number, I add the offset, I get therefore get the physical address and access the physical memory ok. So, as we understand now that this decreases memory needed to store each page table; not each page table to store each page table now we basically have a single page table but it increases time, needed to search the table when a page reference occurs. Now, the way we use the hash table, we can now use a hash table to limit the search to one or a few entries.

So, how do you control the search of this entire space? By using a hash table. So, using this you using this as the hash function using this as a hash function I will I will hash into a particular

place; from where I will get the page frame and I will get into the physical memory. So, by using a hash function I can limit the search to 1 or at most a few page table entries. So, this is how by shifting from a per process page table to one page table for the entire physical memory I can reduce the size of I can reduce the amount of memory dedicated to a whole page tables in main memory.

So, inverted page table is used in many IBM architectures in a few IBM architectures like power pc, etcetera. They are the, they were the first and the main proponents of this inverted page table organization. So, with this we have understood a few ways in which in which page tables are organized and the mechanisms by which um we try to control the size of a page table in main memory.

(Refer Slide Time: 50:14)



We will as a before completing we will take an example and solve a small numerical. A computer uses a computer uses 46 bit virtual addresses, 32 bit physical addresses and three-level page table organization; a three-level page table organization; three-level paged page table organization.

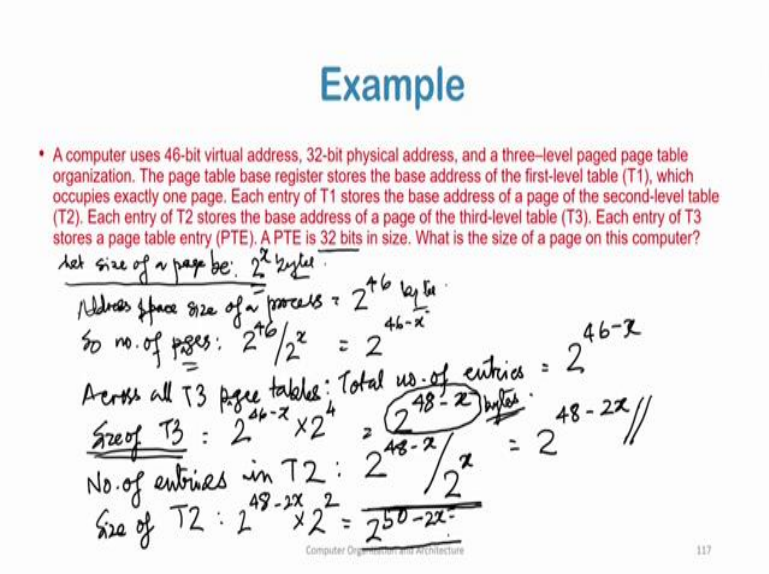
The page table has the page table base register stores the base address of the first level table T1 which occupies exactly one page; this is important. The first level page table occupies exactly one page. Each page each entry of T1 stores base address of a page of the second level. So, each address of T1; so, I have a T1, I have a T1 containing many entries and this T1 holds the

base address of many T2's ok. These are T2's; second level page tables. So, the first level page table which occupies exactly one page; this occupies exactly one page exactly one page.

So, which occupies exactly one page; each entry which occupies exactly one page fine. Each entry of T1 stores the base address of a page of the second level page table, T2. Each entry of T2 stores base address of a page of the third level-page table. Similarly I have a third-level page table T3 ok. So, these are T3's right. So, each entry of T3 stores a page table entry. So, each entry of T3 holds a page table entry. So, this is a page table entry and this basically goes to the physical memory. I have from this page table entry I can find what? I can find the frame number. I can go to the physical memory.

So, each entry of T3 holds a page table entry. A page table entry is 32 bits in size; a page table entry is 32 bits in size; that means, it is 4 bytes or 2^2 bytes so, 32 bits. What is the size of a page on this computer? So, I want to find out the size of a page on this computer.

(Refer Slide Time: 53:13)



So, how do we start? Attack this problem. Let us say that the size of a page be 2^x ok. Now, we have said that the computer has 46 bit virtual addresses. So, what are the total what is the total address space size? The total address space size, address space size of a process what is it equals to; 2^{46} ok.

So, I said that let the size of a page be 2^x. So, number of pages is equal to what? Is equal to $2^{46}/2^x$ because the size of a page is 2^x; that the total address space is 2^{46} bytes; this one is bytes this 2^x bytes. So, therefore, the number of pages is equal to 2^{46-x}.

Now, all these pages so all these pages, so, if I have these pages; that means, I have these many pages in the virtual address space. So, all these pages will have entries in T3. So, combining all they ended with many T3 page tables combining all the T3 page tables what will be the total number of entries in that in the T3 page tables; total combined across all across all T3 page tables, page tables across all T3 page tables total number of entries, total number of entries equals to 2^{46-x} ok.

This will be the total number of entries across all page tables at the T3 level. Now, the size of each entry as I told as we discussed is 32 bits. So, therefore, so therefore which is 4 bytes; so, therefore, what is the size of T3 page tables? Total size total size of T3 page tables; T3 page table, total size of T3 equals to what? $2^{46-x} \times 2^4$ which is equals to 2^{48-x}. This is the total size of T3 page table.

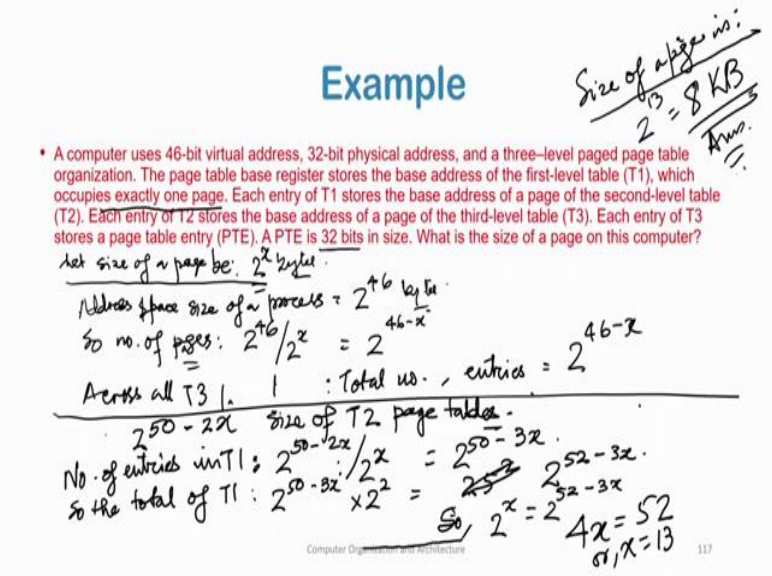
Now, this page table is paged; these page tables are paged. Now so, if this is the total size how many entries corresponding to T3 will be there in T2? So, number of entries number of entries in T2 is given by what? Number of entries in T2 is given by $2^{48-x}/\ 2^x$ which is equals to 2^{48-2x}. Why? Because the size of a page is 2^x; So, therefore, the number this is the total size of the T3 page tables; total size of the T3 page tables in bytes ok.

Now, this will be paged and so, how many pages will be required to store this T3? Assuming that the size of the page remains same; this will be given by $2^{48-x}/\ 2^x$ which is equals to 2^{48-2x} ok. Now, this is the total number of entries.

So, again the size of T2 overall T2's what is it? Assuming that each page table entry is 4 bytes; that will be again given by $2^{48-2x} \times 2^2$ which is equals to 2^{50-2x}.

So, 2^{50-2x} is what? It is the total size of T2 page tables ok. So, basically what am I having?

(Refer Slide Time: 59:03)



So, $2^{50-2x}$is what is the number is the size of is the size of T2 page tables. Now, this will be contained in T1; if this is contained in T1 then number of entries in T1 will again be given by what? $2^{50-2x}/2^x = 2^{50-3x}$.

So, the total total size of T1 is given by $2^{50-2x} \times 2^2 = 2^{50-3x}$. This is the total size of T1 and we had previously said this size is equals to exactly one page. So, $2^x = 2^{50-2x}$. Therefore, what do I get? We get that $4x = 52$ or $x = 13$. So, the size, so the size of a page is; so, the size of a page is what? The size of a page is 2^{13} which is = 8 KB. So, my answer is 8 KB ok.

With this with this we come to the end of this lecture.