Computer Organization and Architecture: A Pedagogical Aspect

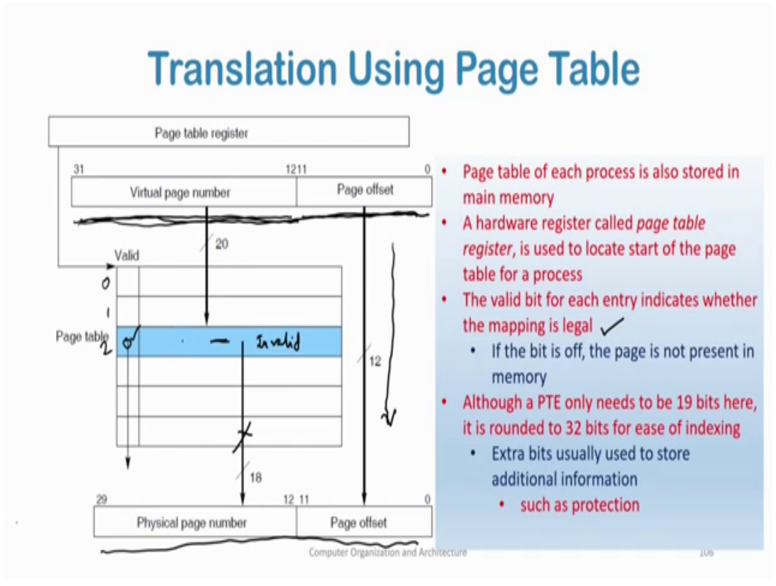
Prof. Jatindra Kr. Deka
Dr. Santosh Biswas
Dr. Arnab Sarkar
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 28
Paging and Segmentation

In this lecture, we continue our discussion with virtual memories. In the last lecture we said that a process generates virtual addresses in order to access memory, a process generates virtual addresses.

Therefore, the CPU generates virtual addresses and to get the required data the this virtual address must be converted to a physical memory address and from that physical memory address the data must be obtained.

(Refer Slide Time: 01:03)



Translation Using Page Table

- Page table of each process is also stored in main memory
- A hardware register called *page table register*, is used to locate start of the page table for a process
- The valid bit for each entry indicates whether the mapping is legal ✓
  - If the bit is off, the page is not present in memory
- Although a PTE only needs to be 19 bits here, it is rounded to 32 bits for ease of indexing
  - Extra bits usually used to store additional information
    - such as protection

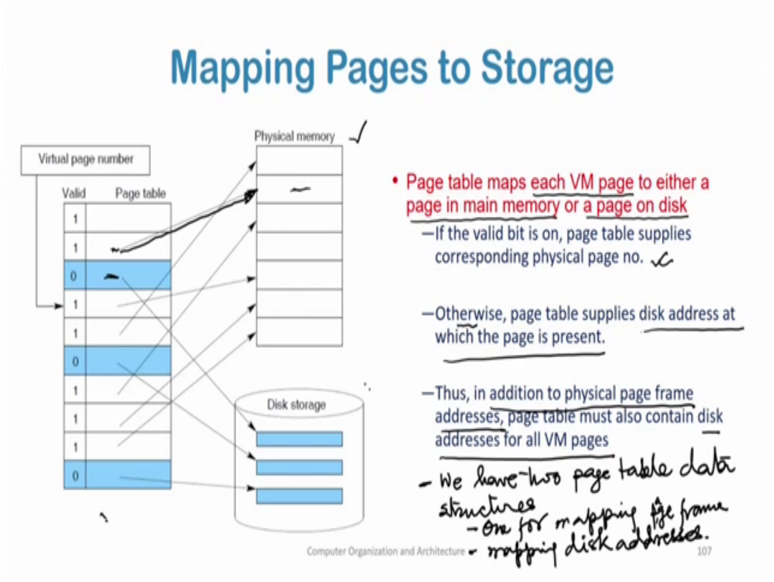
So, we had said that yes. So, this is the virtual address that the CPU generates. This virtual address is divided into two parts: one is the page offset; the other is the virtual page number. The page offset is directly translated and translates translated as it is without any modification for the physical pay for the physical page offset.

This was because the page size in the virtual memory and the page size for the main memory was same is same. The virtual page number is then floated to the page table of the process. So, each process has a page table and this page table is indexed by the virtual page number. So, this is for virtual page number 0, 1, 2. So, this virtual address this virtual address gives the index into the page table entry and in this page table entry if the valid bit is 1, I get the physical page frame number and from this physical page frame number I add the.

So, when I get the physical page frame number and the valid bit is 1 then I add the physical page frame number to the page offset that is directly offset from the virtual address and I generate the complete physical address. This physic this physical address is then floated into the memory address register. And therefore, from the I get the address from in memory from which the data must be accessed.

So, the valid bit what does it tells us? It tells us whether the mapping is legal. If this bit is off what does it mean, if this bit is off the page is not present in the physical memory. So, if the valid bit is 0 0; say suppose this valid bit is 0, this means that this physical page number that this page table entry contains is invalid. This is not valid if this is 0 and therefore, it means that there this, the physical this page is not there in the physical memory and must be brought from the secondary storage.

(Refer Slide Time: 03:44)



Mapping Pages to Storage

- Page table maps each VM page to either a page in main memory or a page on disk
  - If the valid bit is on, page table supplies corresponding physical page no.
  - Otherwise, page table supplies disk address at which the page is present.
  - Thus, in addition to physical page frame addresses, page table must also contain disk addresses for all VM pages
  - We have two page table data structures
    - One for mapping the frame
    - mapping disk addresses.

So, therefore, the page table maps each virtual memory page to either of either a page in main memory or a page on disk. The page table maps each virtual memory page to either a page in main memory or a page on disk. If the valid bit is on, the page table supplies the corresponding physical page frame number. For example, the valid bit is on for this entry and it supplies the physical page frame number from which by which this physical memory can be accessed and this physical memory page frame can be accessed. Otherwise that is if the valid bit is off, the page table supplies the disk address. So, the page table must also contain not only the physical page frame number, it must also contain the disk address of the page.

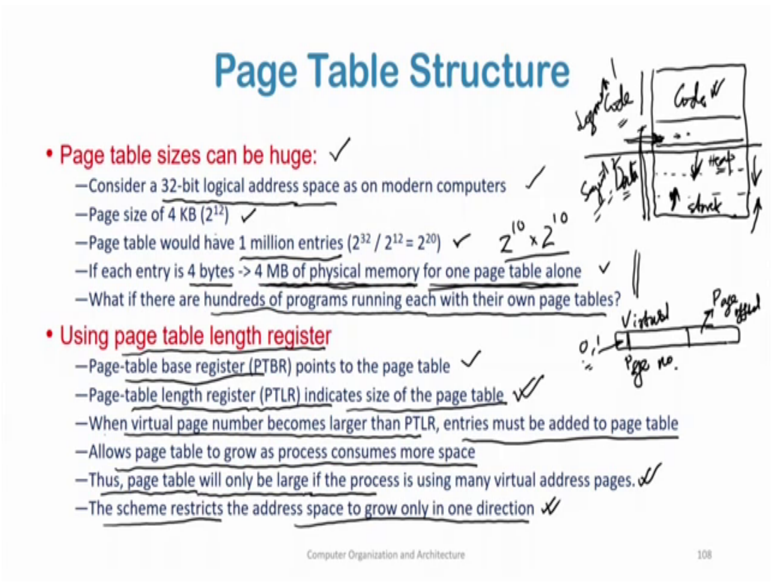
Now, when is this used? For example, even the valid bit is 0 the, this particular page is not present in physical memory. So, this page must be brought from the disk. So, the page table must also contain the disk address of all pages in virtual memory. So, otherwise the page table supplies disk address at which the page is present. Thus, in addition to physical page frame addresses the page table must also contain disk addresses of all virtual memory pages.

That is why although logically same we maintain two distinct, two we distinct two data structures; which both are page tables both are indexed by the virtual addresses, but one for mapping the virtual addresses to secondary storage page addresses and the other for mapping the virtual addresses to the physical memory page frames.

So, we have we have two data structures two page table data structures data structures. One for mapping page frames and the second for mapping disk addresses ok. The other for mapping disk addresses. So, one page table will tell me where in the physical memory this particular virtual page is going to is residing. It may be that this page is not there in physical memory.

Therefore, it is there or not in physical memory it must be there in the secondary storage. So, therefore, for all virtual pages, for all pages in virtual memory corresponding to a process I have a data structure which matches which maps what for corresponding to each of these virtual pages where they are stored in the secondary storage. So, either I have it in physical the pages either in physical memory or in the disk storage.

(Refer Slide Time: 07:20)



Page Table Structure

- Page table sizes can be huge:
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries ($2^{32} / 2^{12} = 2^{20}$)
  - If each entry is 4 bytes -> 4 MB of physical memory for one page table alone
  - What if there are hundreds of programs running each with their own page tables?
- Using page table length register
  - Page-table base register (PTBR) points to the page table
  - Page-table length register (PTLR) indicates size of the page table
  - When virtual page number becomes larger than PTLR, entries must be added to page table
  - Allows page table to grow as process consumes more space
  - Thus, page table will only be large if the process is using many virtual address pages.
  - The scheme restricts the address space to grow only in one direction

Computer Organization and Architecture 108

Now, we come to a discussion; we will now do a discussion on the structure of a page table. Page table sizes can be huge; it can be very big. Let us consider a 32 bit computer. So, which has it has a 32 bit logical address space; 32 bit computers it is a modern computer.

The page size is 4 KB, so therefore 2 to the power 12. Page table would contain the page table if the page size is 4 KB, the page table will contain how many? The page table will contain 1 million entries. Why? The address is 32 bits out of which 12 bits I am using for the page offset; that is why the page size is 2 two to the power 12.

And the rest 32 minus 2 to the power 32 minus 12, these 20 bits I am using for the, for page this 32 bits I am in using for what? The number of entries in the page table so, how many entries will we have in the page table? We will have 1 million entries. Why? 2 to the power 10 into 2 to the power 10 ok; So, I will have 2 to the power 20; so, 2 to the power 20 page table entries.

Now, each page table entries 4 bytes; so, typically page table entries are 32 bits as we discussed. So, therefore, we will have 4 MB of physical memory that will be required for one page table. Now, we have a separate page table for each process. We are seeing that for one process we will require 4 MB of physical and we said that these page tables are resident where these page tables are resident in the physical memory. Yes, these page tables are resident in the physical memory. So, for one process we will consume 4 MB of space just for storing the page table.

Now, if we have now considered the situation where we have 00 s of programs running in; so, we have 00 s of processes running in parallel in the in the system. So, what will be the size of the page tables then? So, 00 programs we are running each consuming 4 MB so, 00 into 4 MB; So, the size in main memory, the amount of main memory consumed for only the page tables is going to be very huge because the size of the page table can be huge. A few techniques are applied in an effort to control the size of the page table.

The one of the simplest technique is as is as follows. So, we use a page table length register, we use a page table length register in order to tell precisely currently what amount of virtual memory space is a process currently taking. So, the page table register which we were discussing previously we will now call it the page table base register and this will as previously will contain the base address of the page table in physical memory. Along with that we will now have a page table length register which will indicate the size of the page table.

Now, the virtual memory of a process can grow with time. So, initially as more and more data as it calls more procedures as it requires for more dynamically allocated memory, so the size of the virtual memory will grow. So, initially the virtual memory of the process will be smaller and then as time passes and it requires more amount of data, it requires more amount of dynamic memory it the process the its virtual address space will grow. So, initially the page table length register will point to the actual size of the virtual address space of the process when it is starting.

So, then what will happen? When virtual page number becomes larger than this PTLR; a page table length register entries must be added to the page table and accordingly the value of the PTLR has to be adjusted. So, therefore, in general the virtual address space of a process will look as look like as follows.

So, I have a code segment and I have a static data segment and I have a dynamic. This whole part is data; this part is code this is the code segment. So, this remains unchanged, this static data segment also remains unchanged; I do not require, but the dynamic data segment it primarily has two parts: one is the stack segment which is used which is allocated as new procedures or functions are called by the program and is and the other is the heap segment from which I allocate dine when I allocate dynamic memory.

So, when I use malloc, calloc I allocated from the heap part of the virtual address space of a process. Now, as more and more data is allocated more and more functions are called, this data portion of the process of the virtual address space of the process is going to increase. So, the number of virtual pages required by the program is going to increase and therefore, the value of the more entries are then needed to be added to the page table of the process and the value of the PTLR has to be also increased.

However, by using this page table length register the advantage is that it allows the page table to grow as process consumes more space. Otherwise what we would have to do? I would have to keep I would have to keeps the page table would otherwise contain space for the entire virtual address space that is possibly addressable by the process. So, if I have a 32 bit computer and I say that 2 to the power 32 is the total logical address space that is addressable by the process then I have to maintain a page table for this entire virtual address space of 2 to the power 32 bytes.

Now, when I have this page table length register I can grow and shrink the page table of the process according to what amount of space the process is actually needing at a given point in time. Thus the page table in this case with the use of a length register will only be as large will only be large, if the process is using many virtual address pages. However, the scheme restricts the address space to grow only in one direction ok. What is the problem that we are saying? The problem that we are trying to indicate is as follows.

As we said this is the code segment and this entire part is the data segment. This is the static part of the data segment which contains the variables, the global variables in the process and I have a stack segment which grows downwards or the heap segment let us say go grows downwards and I have a stack segment which grows upwards.

So, the stack segment grows when more and more I have to allocate activation records for new for newly called functions for nested functions as I call more and more functions; I will add activation records and the stack for the process will grow. This data corresponding to a local function is kept into the stack ok; corresponding to a called function is kept into the stack. From the heap as I told you I allocate dynamic memory using malloc.

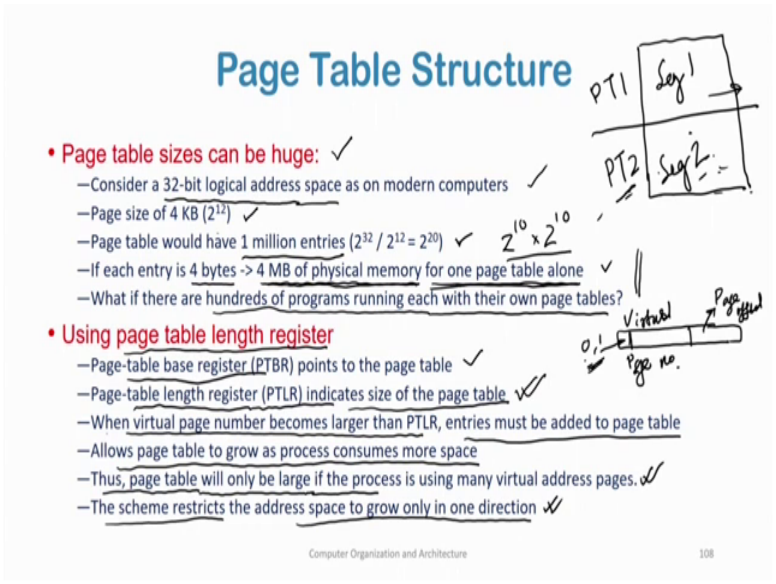
Now, this grows this side and this grows this, this grows on the other side. So, from one side I increase the stack and from the other side I increase the heap. Now, when I have a single page table length register, this obviously, allows this obviously, allows the page table to grow in one direction, but here is the problem.

We are seeing that in typical in typical virtual memory that the virtual memory virtual address space structure of a process that we have it needs to grow in two directions and not only in one direction; To how to solve the problem? The problem can be solved by having let us say I divide this address space into two parts.

So, I keep the let us say that I said that the virtual memory previously had what? The virtual memory had a virtual this was the page offset part; page offset part and this was the virtual page number ok, virtual page number. Now, this virtual page number we will keep the most significant bit to indicate to 0 it can be 0 or 1 and it will be divided into two parts. So, by using this one I divide the virtual address space into two parts. Now, out when I divide the virtual address space into two parts and I name it say segment 1 and this one segment 1 and this one as segment 2.

And I use a separate page table for each segment. I use a separate page table for each segment. So, therefore, this bit will tell me whether to look for page table whether to look into the page table of segment 1 or to look into the page table of segment 2? Now, depending on which segment I am using. So, and each of these page tables will have its own length register. So, the situation if we if we if you want to say it again is something like this.

(Refer Slide Time: 18:28)



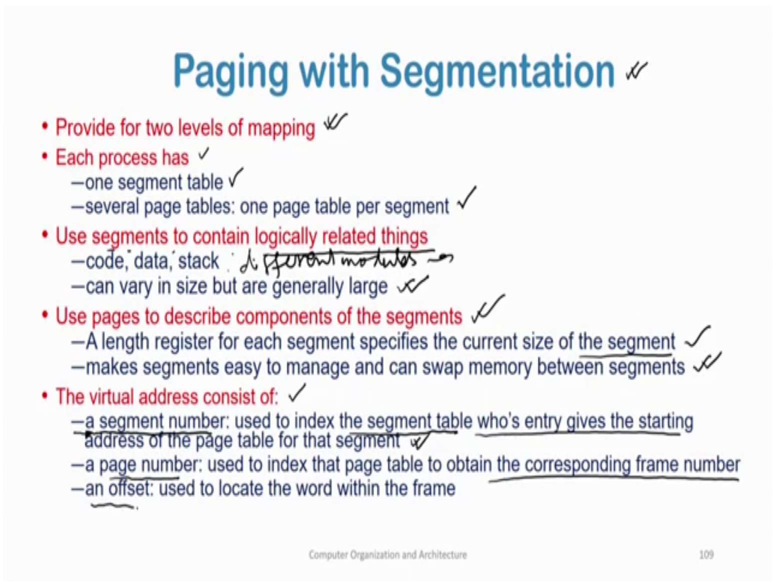
Page Table Structure

- Page table sizes can be huge:
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries ($2^{32} / 2^{12} = 2^{20}$)
  - If each entry is 4 bytes -> 4 MB of physical memory for one page table alone
  - What if there are hundreds of programs running each with their own page tables?
- Using page table length register
  - Page-table base register (PTBR) points to the page table
  - Page-table length register (PTLR) indicates size of the page table
  - When virtual page number becomes larger than PTLR, entries must be added to page table
  - Allows page table to grow as process consumes more space
  - Thus, page table will only be large if the process is using many virtual address pages.
  - The scheme restricts the address space to grow only in one direction

So, we have this virtual address space. This virtual address space is divided into two parts and this division is based on the most significant bit in the virtual page number part of the virtual address and this I will called segment 1, this I will call segment 2 and I will have a separate page table for segment 1 and I will have a separate page table for segment 2.

Now, whether I will use page table 1 or page table 2 will be given by this most significant bit. Each of this page table will have a separate length register. So, each of these page tables they can then grow separately ok. So, this problem of this restriction that the page table can grow only in one direction is solved by having two page tables for the process for corresponding to its two segments.

(Refer Slide Time: 19:28)


Paging with Segmentation

- Provide for two levels of mapping
- Each process has
  - one segment table
  - several page tables: one page table per segment
- Use segments to contain logically related things
  - code, data, stack : different modules
  - can vary in size but are generally large
- Use pages to describe components of the segments
  - A length register for each segment specifies the current size of the segment
  - makes segments easy to manage and can swap memory between segments
- The virtual address consist of:
  - a segment number: used to index the segment table who's entry gives the starting address of the page table for that segment
  - a page number: used to index that page table to obtain the corresponding frame number
  - an offset: used to locate the word within the frame

Computer Organization and Architecture 109

Now, the even if we have two segments the problem is that if the virtual memory or the virtual address space of the process is scattered; it has different types of modules and each module is in a different part of the address space the usage is not that good. Even we have a length register which restricts the use, if the virtual address space is scattered we understand that that the length register has to be big, but a let the length register has a certain value.

But within that length register many of the virtual many of the virtual pages basically do not map to actual logical addresses that is used by the process. So, the virtual pages that are used by the process is scattered. If it is scattered even if we have two page tables and two page, page table length registers it the usage the page table will still be large and the usage is not as efficient.

To handle this problem we sometimes use paging with segmentation. This segmentation is a little different from the segmentation we just discussed in the previous slide. So, this also provides two levels of mapping but this segmentation is a bit more flexible. Each process has a segment table and several page tables one page tables for each segment.

This is also it is very similar to what we discussed last in the last slide. What is different comes now. We use segments to contain logically related things. Now for code, data stack different modules different modules sorry different modules we use separate segments. So, for each for each of the different modules that are required each of the libraries that are required by this process we will use a separate segment and the segment size can vary but are they are generally large. Each of these segments again have again have a length register a segments can have different lengths. It will have it and each of these segments will be paged.

So, we use pages to describe components of the segments. A length register for each segment specifies the current size of the segment as we just said. This makes segments easy to manage and can and can swap memory between segments. We can also allow swapping of memory between segments.

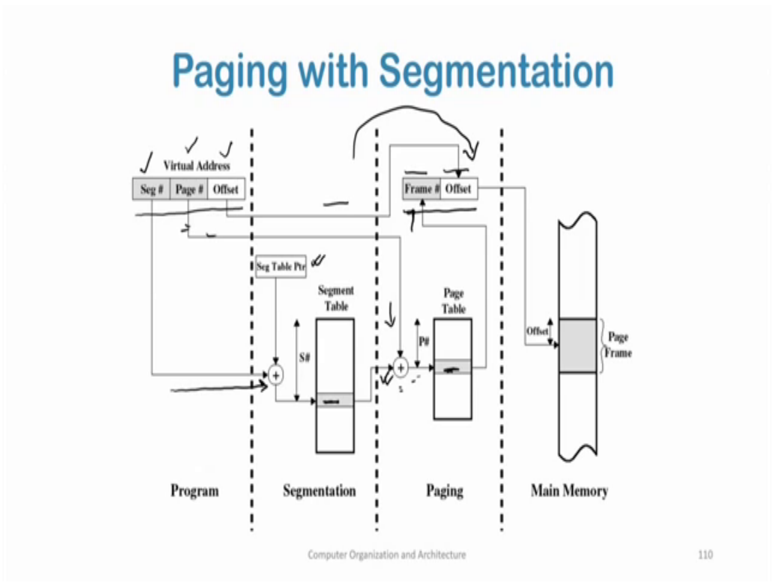
We will possibly see this technique later in our in a later lecture. So, the virtual address now consists of what a segment number. So, previously the virtual a virtual address was divided into two parts one page offset and one virtual page number. Now it will be divided into three parts. It will first have a segment number.

The segment number will be used to index the segment table whose entry gives the starting address of the page table for that segment. So, whose entry gives the starting address of the page table for that segment. So, I have divided my so I have divided my process into a number of segments and each such segment now has a separate page table.

So, a segment number will tell me will point to the starting address of the page table corresponding to that segment. A page number now, the in that page table I will go I will index that page table using the page number. So, I get into a segment, I go to the start of the page table corresponding to that segment, then I use the page number part of the virtual address to index which index a page into the page table.

So, it is used to index that page table to obtain the corresponding frame number corresponding frame number. Now, when I get into the page table, the page table entry tells me which physical page frame um where which physical page frame contains my data corresponding to the virtual page number that I have. Now, then the offset tells me as before where within that physical page frame is my required data.

(Refer Slide Time: 24:15)



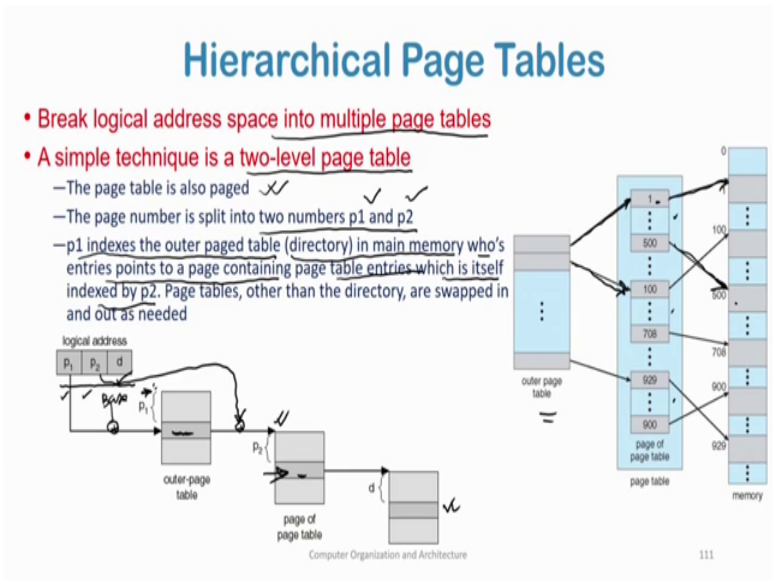
Paging with Segmentation

So, therefore, the virtual address now consists of three parts: segment number page number and page offset. I go I take the segment number and then from the segment table pointer this will tell me the start of my segment table in main memory and I index into this segment table using the segment number and I get to the particular segment that is from which I need the data; So, which module if the if the segment indicates a module.

So, I go to the appropriate segment and in that module ok. So, I get the segment or that module. So, in that segment what do I do from that segment I go to what do I get? I get the starting address of the page table; I get the starting address of the page table corresponding to that segment I get the starting address of the page table. And the index into that page table index where within that page table I have to go is given by my page number part of the virtual address. When I get this two, I add this two up and I get to the page table entry.

When I get to the page table entry from here I get the physical page frame number. When I get the physical page frame number I added to the offset part which directly comes from here from the virtual address and I add I add the frame part and the offset part to get the total physical address and from that physical address I basically go to the main memory and get my data. I get the page frame and from that page frame I get the data required data.

(Refer Slide Time: 26:03)



Hierarchical Page Tables

- Break logical address space into multiple page tables
- A simple technique is a two-level page table
  - The page table is also paged
  - The page number is split into two numbers p1 and p2
  - p1 indexes the outer paged table (directory) in main memory who's entries points to a page containing page table entries which is itself indexed by p2. Page tables, other than the directory, are swapped in and out as needed

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 in 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 um 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 a 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 do not 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 levels page table. So, what happens in a two levels 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.

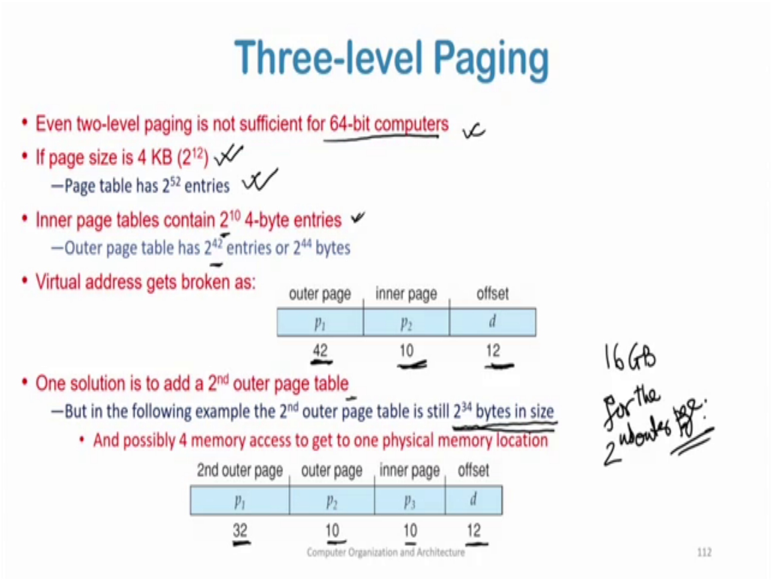
So, it this allows the page table to be paged and therefore bit better sharing of the main memory is possible. The page number is split into two; the page number is now split into two parts. One is p1; the other is p2; p1 indexes the outer page table; p1 indexes the outer page table or directory in main memory whose entries point to a page containing the page table entries which itself is indexed by p2.

So, what happens? The logical address space is now divided into two parts; p1 p2 and this is the offset part. So, from p1 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 p1 that I have I get into where in the outer page table is the start in memory of the second page table p2. So, start in main memory for the second page table p2 is obtained in the entry for the outer page table. From here I from here I use this one from I now add this p2; I add this p2 with this value that I get and I get this one and I get this one where in p2 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)



Three-level Paging

- Even two-level paging is not sufficient for 64-bit computers ✓
- If page size is 4 KB (2^{12}) ✓
  - Page table has 2^{52} entries ✓
- Inner page tables contain 2^{10} 4-byte entries ✓
  - Outer page table has 2^{42} entries or 2^{44} bytes
- Virtual address gets broken as:

outer page	inner page	offset
p_1	p_2	d
42	10	12

- One solution is to add a 2nd outer page table
  - But in the following example the 2nd outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

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

16GB

for the 2^{nd} outer page.

Computer Organization and Architecture

112

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 to the power 52 entries ok. So, 2 to the power 64 minus 2 to the power minus 12, 2 to the power 64 minus 12 equals to 2 to the power 52.

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

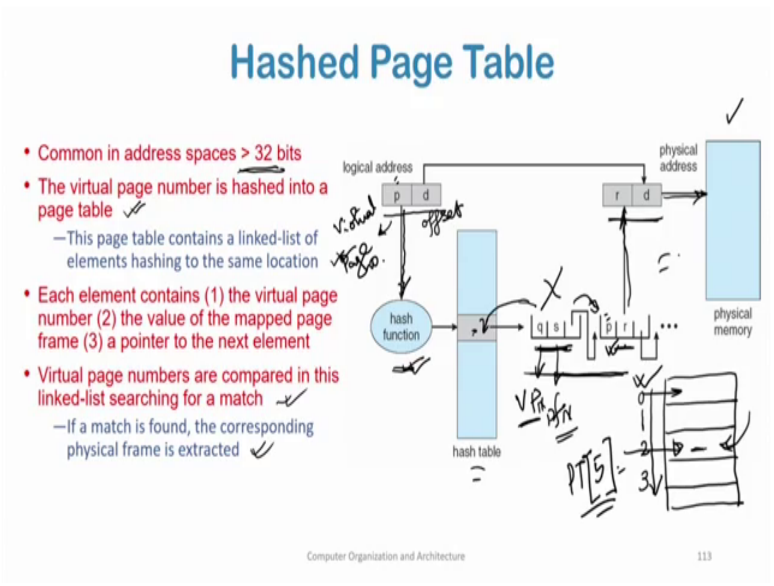
So, my offset contained this 12 bits, so 2 to the power 12 12 bits. The inner page table contains what? 2 2 to the power 10 entries; So, I use 10 bits. So, now, my outer page table has 2 to the power 42 entries. So, this one is still 42 bits. So, it contains 2 to the power 42 entries right.

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 to the power 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? 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 to the power 34 bytes in size. Why? Because I have 2 to the power 32 entries and each such entry is 4 bytes or 2 to the power 2 bytes. So, the size of the outer second order outer page table is 2 to the power 34 bytes which is very huge. We understand that 2 to the power 30 means 1 GB. So, if the therefore, 2 to the power 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 a 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)



Hashed Page Table

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - –This page table contains a linked-list 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 linked-list searching for a match
  - –If a match is found, the corresponding physical frame is extracted

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 pay 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 an and get to a location um 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 pay 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 added 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 have to the I have 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 do not 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 to a 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 no; I keep an entry for all for all at a virtual pages that I have and then that makes it easy to search for a particular page table entry.

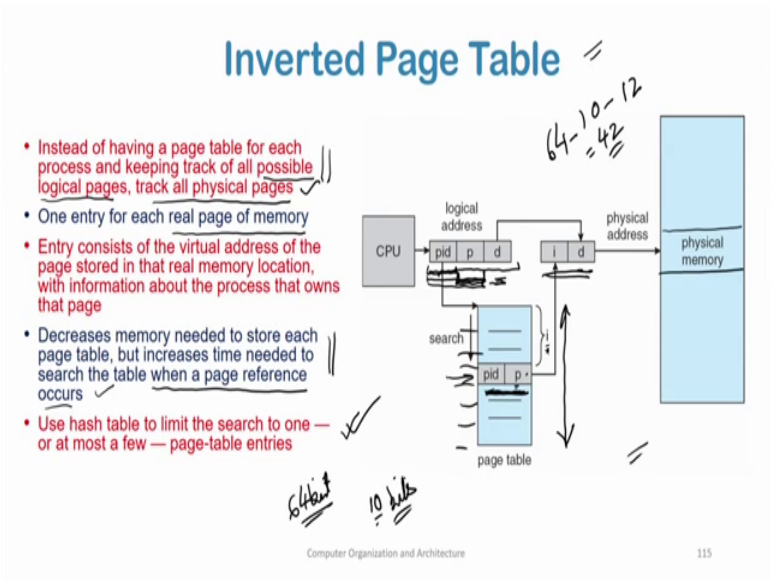
Because now, if I am search if I want to search for logical address numbers, a logical page number 5; virtual page number 5, I know that it is it can be indexed by page table 5. I can I can just go to page table 5 and I will get this entry ok.

But, but then this will not be possible if I do not 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 the 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 do not 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)



Inverted Page Table

- Instead of having a page table for each process and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

Computer Organization and Architecture 115

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 happens 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 vary. 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 a 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 save 10 bits for my process id. So, if I have a 64 bit system, 64 bit logical address I still have 64 minus 10 minus 12 which is equals to 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.

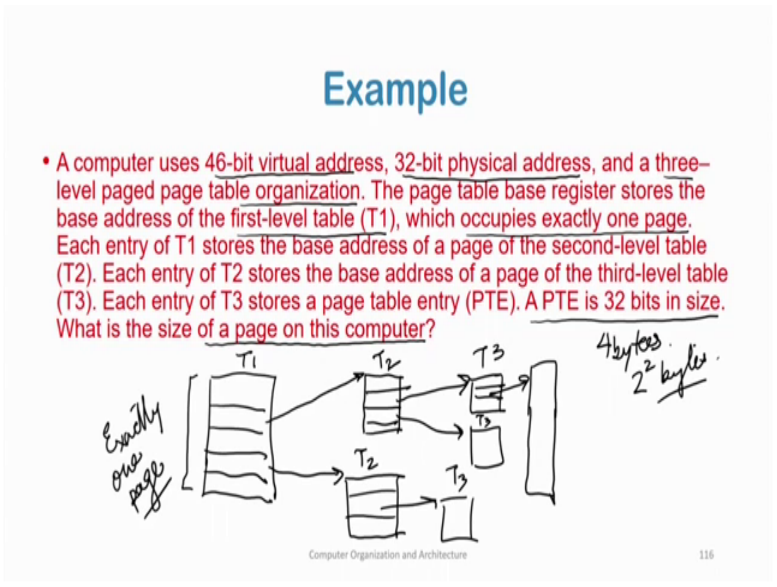
If I get this match, if I get this match I know wherein the page table I got this match which is i; where the index to the page table where I got the match. And this index basically tells me the physical page frame number.

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, but 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 of you 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 inverter 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)



Example

- A computer uses 46-bit virtual address, 32-bit physical address, and a three-level paged page table organization. The page table base register stores the base address of the first-level table (T1), which occupies exactly one page. Each entry of T1 stores the base address of a page of the second-level table (T2). Each entry of T2 stores the base address of a page of the third-level table (T3). Each entry of T3 stores a page table entry (PTE). A PTE is 32 bits in size. What is the size of a page on this computer?

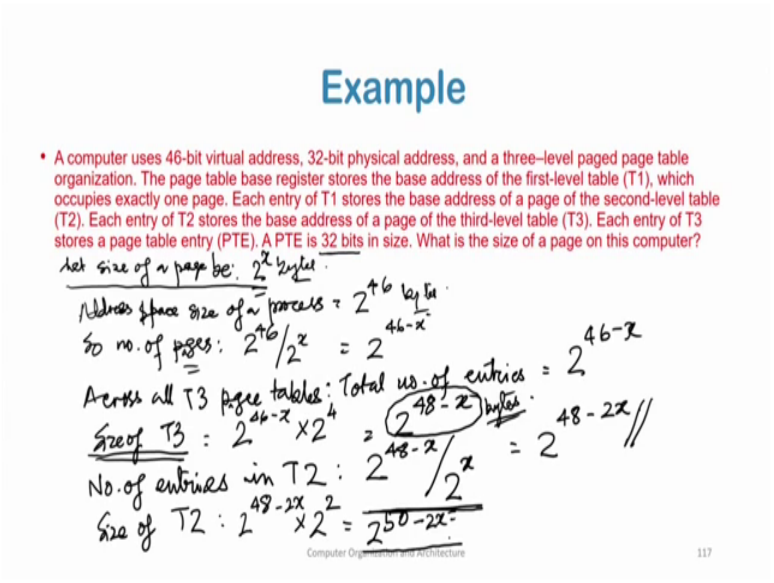
We will as a before completing we will take an example and solve a small numerical. A computer uses a computer uses 46 bit virtual addresses, 32 bit physical addresses and three-level page table organization; a three-level page table organization; three-level paged page table organization.

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 2 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 to the power 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)



Example

- A computer uses 46-bit virtual address, 32-bit physical address, and a three–level paged page table organization. The page table base register stores the base address of the first-level table (T1), which occupies exactly one page. Each entry of T1 stores the base address of a page of the second-level table (T2). Each entry of T2 stores the base address of a page of the third-level table (T3). Each entry of T3 stores a page table entry (PTE). A PTE is 32 bits in size. What is the size of a page on this computer?

Let size of a page be: 2^x bytes.

Address space size of a process = 2^{46} bytes.

So no. of pages: $2^{46}/2^x = 2^{46-x}$

Across all T3 page tables: Total no. of entries = 2^{46-x}

Size of T3 : $2^{46-x} \times 2^4 = 2^{48-x}$ bytes.

No. of entries in T2 : $2^{48-x}/2^x = 2^{48-2x}$

Size of T2 : $2^{48-2x} \times 2^2 = 2^{50-2x}$.

Computer Organization and Architecture 117

So, how do we start? Attack this problem. Let us say that let the size of a page be 2 to the power x, the size of a page be 2 to the power 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 to the power 46 ok.

So, I said that let the size of a page be 2 to the power x. So, number so, number of pages. So, number of pages is equal to what? Is equal to 2 to the power 46 by 2 to the power x because the size of a page is 2 to the power x; that the at total address space is 2 to the power 46 bytes; this one is bytes this 2 to the power x bytes. So, therefore, the number of pages is equal to 2 to the power 46 minus 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 p T3 page tables what will be the total number of entries in that in the t T3 page tables; total combined across all across all T3 page tables, page tables across all p T3 page tables total number of entries, total number of entries equals to 2 to the power 46 minus 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 to the power 46 minus x into 2 to the power 4 which is equals to 2 to the power 48 minus x. This is the total size of T3 page table or this is the total size of the T3 page tables total size of the T3 page tables.

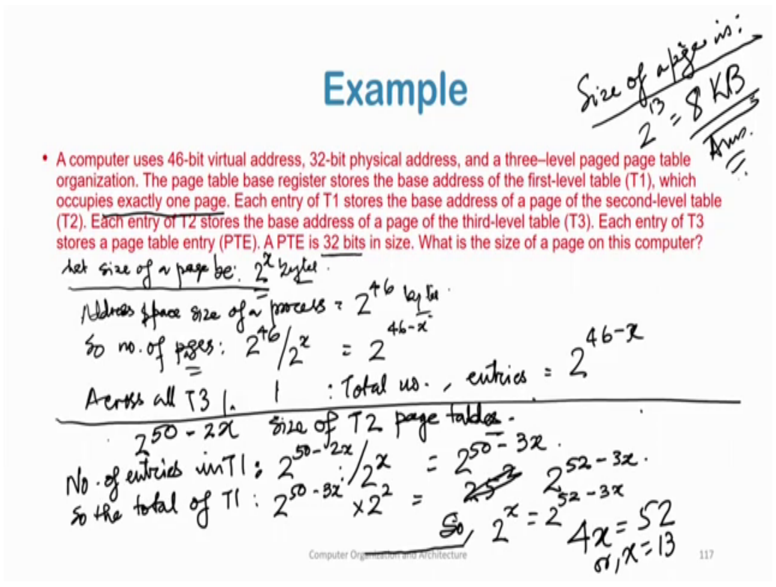
Now, this page table is paged; these page tables are paged. Now so, if this is the total size how many entries corresponding to T3 will be there in T2? So, number of entries number of entries in T2 is given by what? Number of entries in T2 is given by 2 to the power 48 minus x divided by 2 to the power x which is equals to 2 to the power 48 minus 2x. Why? Because the size of a page is 2 to the power 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 to the power 48 minus x divided by 2 to the power x which is equals to 2 to the power 48 minus 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 to the power 48 ok; minus 2 x into 2 to the power 2 which is equals to 2 to the power 2 2 which is equals to what; 2 to the power 5 0, 2 to the power 50 minus 2x.

So, 2 to the power 50 minus 2x is what? It is the total size of T2 page tables ok. If I rub this, if I if I rub this to get of some white space if I rub this. So, basically what am I having?

(Refer Slide Time: 59:03)



Example

- A computer uses 46-bit virtual address, 32-bit physical address, and a three–level paged page table organization. The page table base register stores the base address of the first-level table (T1), which occupies exactly one page. Each entry of T1 stores the base address of a page of the second-level table (T2). Each entry of T2 stores the base address of a page of the third-level table (T3). Each entry of T3 stores a page table entry (PTE). A PTE is 32 bits in size. What is the size of a page on this computer?

Let size of a page be: 2^x bytes.

Address space size of a process $= 2^{46}$ bytes.

So no. of pages: $2^{46}/2^x = 2^{46-x}$

Across all T3 | : Total no. of entries $= 2^{46-x}$

2^{50-2x} size of T2 page tables.

No. of entries in T1: $2^{50-2x}/2^x = 2^{50-3x}$.

So the total of T1: $2^{50-3x} \times 2^2 = \cancel{2^{52}}\ 2^{52-3x}$.

So, $2^x = 2^{52-3x}$

$4x = 52$

or, $x = 13$

Size of a page is: $2^{13} = 8$ KB Ans.

Computer Organization and Architecture 117

So, 2 to the power 2 to the power 50 minus 2x; 2 to the power 50 minus 2x is what is the number is a size of is the size of T2 page tables; is the size of T2 page tables ok; 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 2 to the; So, number of entries in T1 is given by what? 2 to the power 50 minus 2x divided by 2 to the power x equals to 2 to the power 50 minus 3x.

So, the total so, that so, the so, the total size of T1 is given by 2 to the power um. So, total size of T1 is given by 2 to the power 50 minus 3x into 2 to the power 2 is equals to 2 to the power 52 2 to the sorry, 2 to the power 52 minus 3x. This is the total size of T1 and we said that this size is equal to exactly one page.

We had previously said this size is equals to exactly one page. So, 2 to the power x, so 2 to the power x is equals to 2 to the power 52 minus 3x. Therefore, what do I get? We get that 4x equals to 52 or x or x equals to 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 to the power 13 which is equals to 8 KB. So, my answer is 8 KB ok.

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