

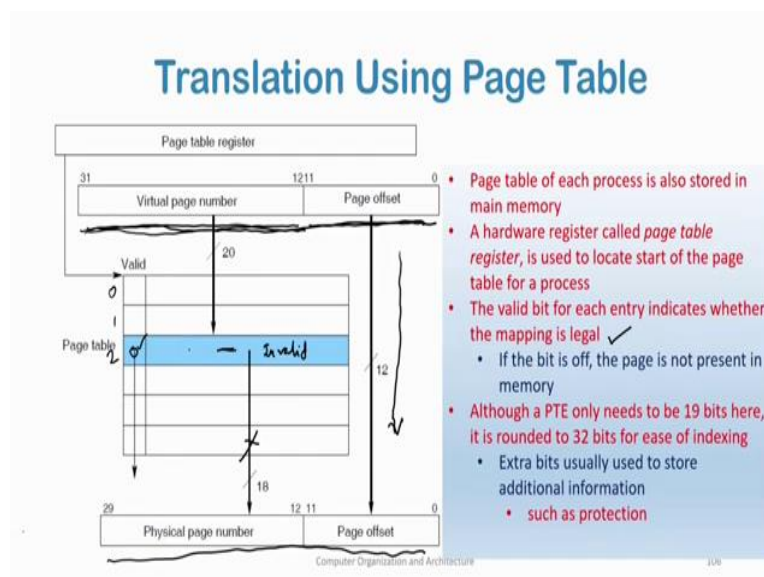
**Computer Organization and Architecture: A Pedagogical Aspect**  
**Prof. Jatindra Kr. Deka**  
**Dr. Santosh Biswas**  
**Dr. Arnab Sarkar**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology, Guwahati**

**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 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)



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 page for the physical page offset.

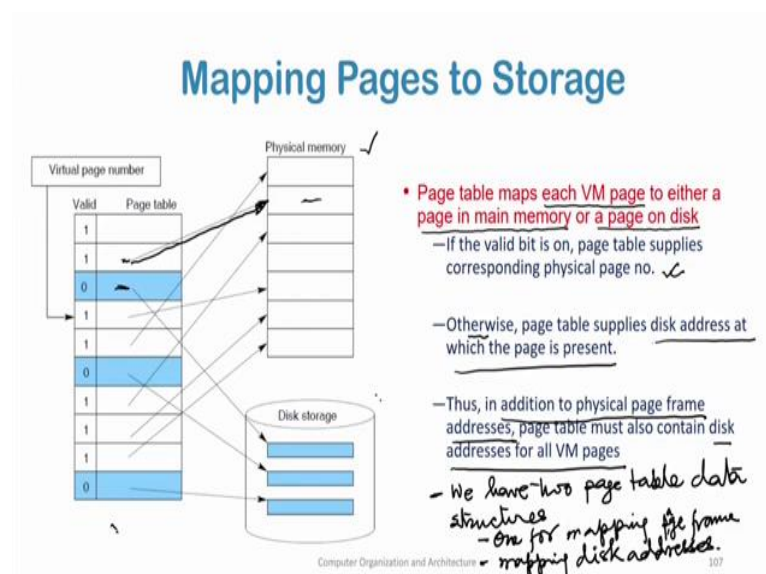
This was because the page size in the virtual memory and the page size for the main memory was same is same. The virtual page number is then floated to the page table of the process. So,

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 physical address is then floated into the memory address register. And therefore, from I get the address 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 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 this page is not there in the physical memory and must be brought from the secondary storage.

(Refer Slide Time: 03:44)



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.

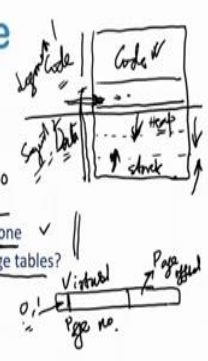
So, we have we have two data structures two page table data structures data structures. One for mapping page frames and the second for mapping disk addresses ok. The other for mapping disk addresses. So, one page table will tell me where in the physical memory this particular virtual page is going to 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 ( $2^{32} / 2^{12} = 2^{20}$ ) ✓  $2^{10} \times 2^{10}$
  - If each entry is 4 bytes  $\rightarrow$  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

308

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 its a modern computer.

The page size is 4 KB, so therefore  $2^{12}$ . 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  $2^{12}$ .

And the rest 32 minus 12 these 20 bits I am using for the, for page this 32 bits I am 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?  $2^{10} \times 2^{10}$  ok; So, I will have  $2^{20}$  page table entries.

Now, each page table entry is 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 saying 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 hundreds of programs running in; so, we have hundreds of processes running in parallel in the in the system. So, what will be the size of the page tables then? So, 100 programs we are running each consuming 4 MB so,  $100 \times 4 \text{ MB}$ ; 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 ok 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 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 don't 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 when I allocate dynamic memory.

So, when I use malloc, calloc I allocate 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 contains 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  $2^{32}$  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  $2^{32}$  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. The 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 ok 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 saying 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; So 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 see 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 ( $2^{32} / 2^{12} = 2^{20}$ ) ✓  $2^{10} \times 2^{10}$
  - If each entry is 4 bytes  $\rightarrow$  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 ✓

The diagram shows two page tables, PT1 and PT2, labeled 'Seg 1' and 'Seg 2' respectively. Below them, a virtual address is shown as a horizontal bar divided into two parts: 'Page no.' (Virtual) and 'Page offset' (Physical). The 'Page no.' part is further divided into two segments by a bit (0/1).

Computer Organization and Architecture 108

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 call 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 its 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 they are generally large. Each of these segments again have again have a length register, 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.

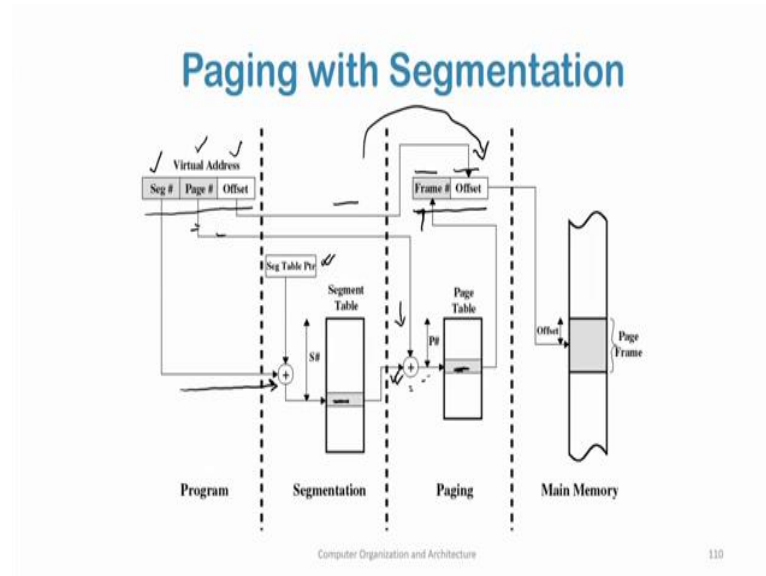
We will possibly see this technique later in our in a later lecture. So, the virtual address now consists of what a segment number. So, previously the virtual a virtual address was divided into two parts one page offset and one virtual page number. Now it will be divided into three parts. It will first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



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 ok 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.