

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

Lecture - 48
Memory Management (Contd.)

Next, we look into another, the memory page table organization strategy, which is known as inverted page table.

(Refer Slide Time: 00:31)

The slide is titled "Inverted Page Table" and contains the following text:

- Rather than each process having a page table and keeping track of all possible logical pages, track all the physical pages
- Use **inverted page-table**, which has one entry for each real page of memory
- An entry the inverted-page table consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- What is maximum size of the inverted page-table?

Handwritten notes on the slide include: $P_1, P_2, P_3, \dots, P_{100}$ at the top right; a circle containing P_1, P_2, P_{10} with an arrow pointing to the first entry of a vertical list of memory slots; and the word "memory" written at the bottom of the list. The list has two entries with arrows pointing to them labeled P_{k1}, P_{g2} and P_{k2}, P_{g6} .

The idea is that see, at any point of time if you look into the system. So, there are large number of processes and each process has got it is own page table. Now, even if the process is not running, if it is ready to run state or in the blocked state something like that, so, we have to, have it is corresponding page table stored in the main memory.

So, that is the problem. So, unnecessarily we are blocking a large number or large amount of memory keeping the page table, but ultimately what is happening is in that, is that in the memory frame whichever processes are there. So, they are whichever process pages are fitting. So, they are only coming into picture.

So, it may so, happen that I have got all these processes P 1, P 2, P 3 up to say P 100, there are 100 processes that are currently there in the system. But at present we have, we are bothered about only a few processes say P 1 P 2 and P 10, but it should be sufficient.

So, if I just remember the page tables corresponding to P 1, P 2, P 10 and it is not required to remember the page table of all other processes.

So, can we do something so that, I just remember the information related to this three pages only and do not have the others. Or in other words I can say that, if this is the memory structure, if this is the memory and this memory is organized in terms of frames, it is divided into frames. So, I need to remember for each frame what the page that it is holding here is. Now a page can be identified by the process of which this page belongs to and the corresponding. So, if it may be, that this is process 1 page number 2, this frame may be corresponding to process 2 page number 6.

So, like that if we remember, so I have to remember information only for the amount of information equal to number of frames that we have in the physical memory. So, we are not storing any additional information the spanning over all the processes. So, we are storing it frame wise, instead of storing it process wise, we are storing the page table frame wise.

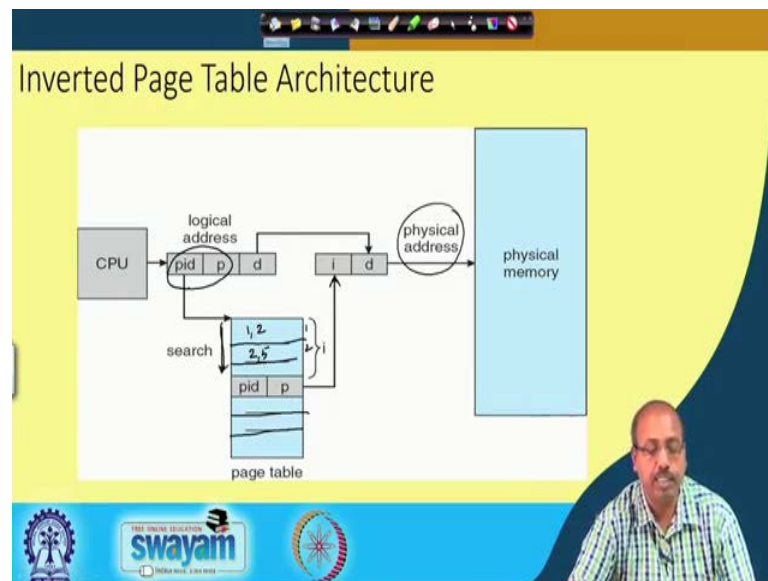
So, this is the idea of inverted page table. So, rather than each process having a page table and keeping track of all possible logical pages, we track all the physical pages. So, instead of tracking the logical pages of processes, we track the physical pages of the processes.

Now of the system, now, this structure gives rise to inverted page table, so, which has one entry for each real page of memory. So whatever, whichever pages have been loaded into the memory for them only, I will have an entry in the inverted page table. So, inverted page table entry it consists of the virtual address of the page stored in that real memory location with information about the process that owns that page. As I was telling for every frame, now I need to remember it is from which process and which page of that process.

So, that way we have to remember these two information and they are remembered like this. So, we have go for each actual physical frame. So, we store the, corresponding process the from which this page is coming and the corresponding page of that process which is coming here. So, this an entry in the inverted page table it consists of the virtual address of the page stored in that real memory location with information about the process that owns that page.

And then what is the maximum size of inverted page table? The answer is definitely equal to number of memory frames that we have in this that is allowed in the system. So, if the if there are 100 frames then this page table size is becoming equal to 100. Now advantages, definitely the space that we have. So, space requirement reduces, I need to remember only this thing.

(Refer Slide Time: 04:43)



Now what is happening is that for a particular when an address is generated by CPU the logical address. So, it is considered at the pid the process identifier has been appended to the logical address. So, the CPU generates this p and d.

Now, from the running process information. So, you can append this pid part from there. So, this logical address consists of three components, now pid, p and d. And then we have to search for this pid and p into this inverted page table. So, inverted page table, we search for this entry pid and p. So that, way when I come to, this particular entry. So I have got this it is at a distance i from the beginning so; that means, that this is the i th frame. So, for every frame I am remembering. So, this thing is. So, this is for frame 1, this is for frame 2. So, like that it is going and this is for the last frame.

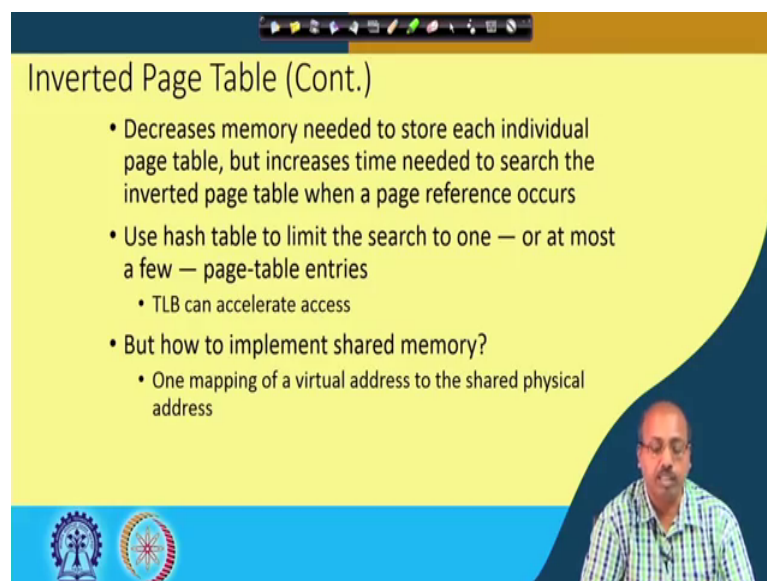
So, if frame 1 is having the information for process 1, page number two this is frame 1. Frame 2 is having information about, say process 2 page number 5. So, like that it is going. So that way, if I am, if I find this match of this pid, p combination at index i at index i; that means, the corresponding frame number is i. So, this i is taken here and it is

made the we get the physical frame number i and this displacement comes and that gives me the physical address.

So, that is how this inverted page table works, the advantage is that one, but the difficulty that we have is searching into this page table, because now you cannot have any other search method. So, you have to look from the beginning and search for this pid p combination of course, you can do something better by having some hash and all. So, if you can do a hash function calculation and then that may be utilized, but that is again cumbersome because there may be collision and all. So, all those things have to be resolved.

But somehow whatever we do ultimately we have to locate this pid p combination into the page table and if that pid p combination is residing at entry i of this page table, then i is the corresponding frame number. So, this is how we get the physical address for a logical address.

(Refer Slide Time: 07:21)



The slide is titled "Inverted Page Table (Cont.)" and features a yellow background with a blue wave-like shape on the right side. At the bottom left, there are two circular logos. At the bottom right, there is a small video inset showing a man in a green and white checkered shirt speaking. The slide contains the following text:

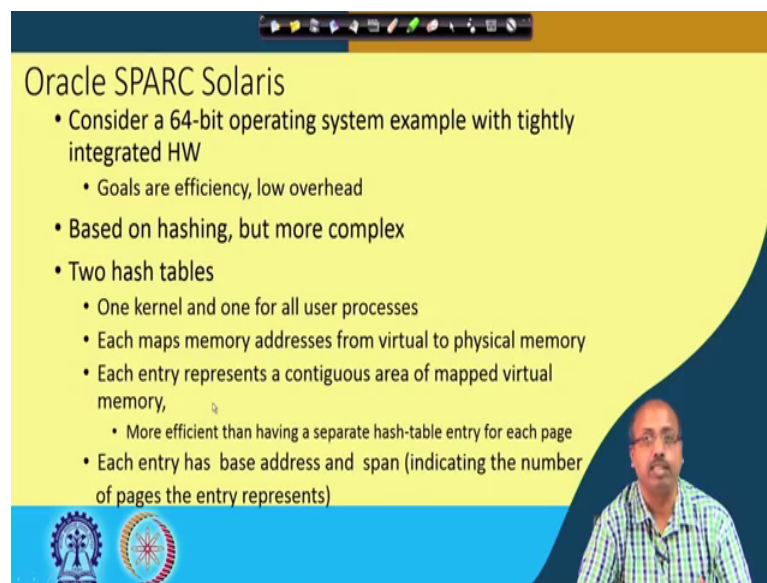
- Decreases memory needed to store each individual page table, but increases time needed to search the inverted page table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
 - TLB can accelerate access
- But how to implement shared memory?
 - One mapping of a virtual address to the shared physical address

So, it decreases the memory needed to store each individual page table, because we do not have to store this page tables for individual processes. But increases the time needed to search the inverted page table, when a page reference occurs. So, you have to search into that list, most probably sequentially search. So, we can use some hash table to limit the search to one or at most a few page table entries. So, that is one possibility. So, now, we can use a as I was telling that we can use the hashing function for that.

Other possibility is that we store a part of this one in to the TLB. Now you see this since, this size is fixed. So, this is equal to number of frames that we have in the memory. So, if you can keep this entire inverted page table in the TLB, translation look aside buffer that is the that will do parallel search for this pid p into all these locations concurrently and if it can find a match. So, then it will come up with that index.

So, this way we can do this thing. So, either we can use a hash table to limit the search to one or at most a few entries or we can use a TLB to accelerate the access. But implementing shared memory is a problem. Now you see that, one mapping of a virtual address with the shared physical address so, that is difficult. Because now, a number of processes pages they want to share the same memory frame so, that is difficult to implement. But anyway, everything has got a positive and negative. So, the frozen cons that has to be taken.

(Refer Slide Time: 08:57)



The slide is titled "Oracle SPARC Solaris" and contains the following text:

- Consider a 64-bit operating system example with tightly integrated HW
 - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
 - One kernel and one for all user processes
 - Each maps memory addresses from virtual to physical memory
 - Each entry represents a contiguous area of mapped virtual memory,
 - More efficient than having a separate hash-table entry for each page
 - Each entry has base address and span (indicating the number of pages the entry represents)

The slide also features a video feed of a presenter in the bottom right corner and two logos in the bottom left corner.

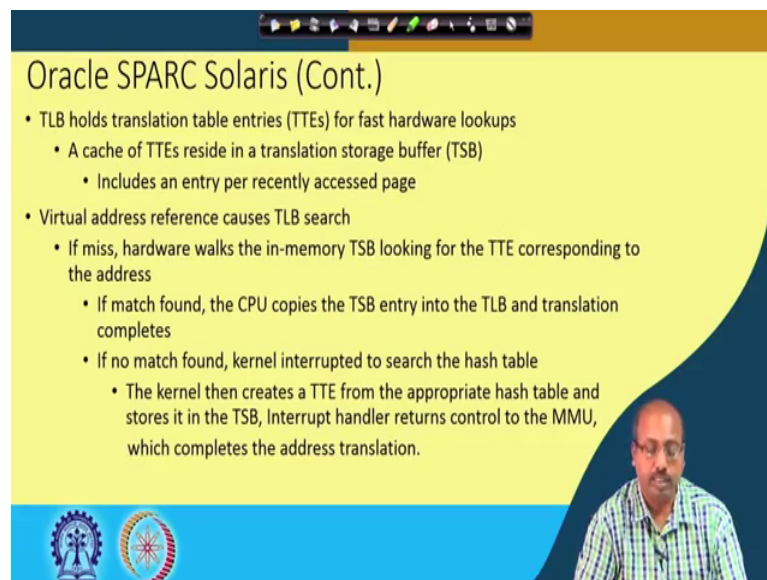
Then considering, some of the operating systems that use this paging scheme, one is Oracle SPARC Solaris consider a 64 bit operating system, example with tightly integrated hardware. So, goals are efficiency and a low overhead, it is based on hashing, but it is more complex, in the sense that there are two hash tables, one at the for the kernel and one for all the user processes.

So, with kernel hash table, so that way, kernel the pages more or less fixed. So, they are using this part and then for other user so, there is another hash table. So, each of the map

has a memory addresses from virtual to physical address and each entry represents a contiguous area of mapped virtual memory. So that is true for any hash based page table organization.

So, it is more efficient than having a separate hash table entry for each page, because we have got; we have got a this thing kernel and users hash tables are different. And each entry has got a base address and a span indicating the number of page, pages that the entry represents. So, we have got a contiguous area of mapped virtual memory. So, that way the page table page size can be a variable.

(Refer Slide Time: 10:18)



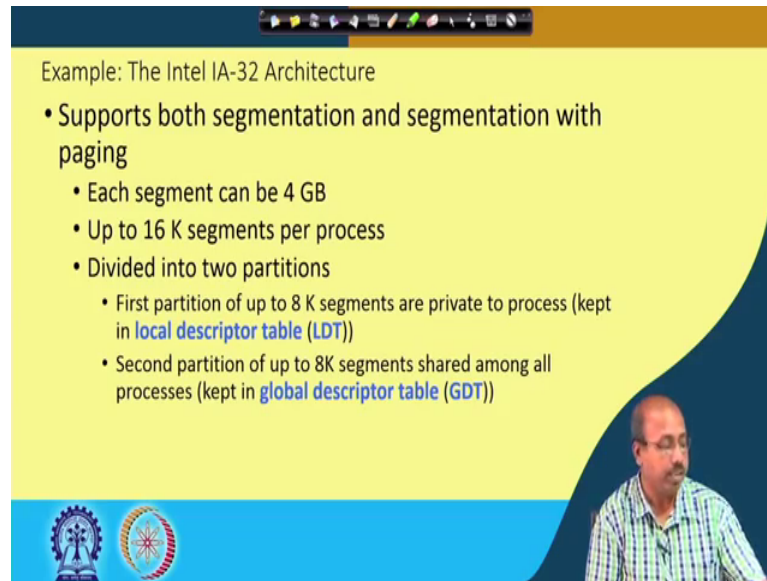
Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
 - A cache of TTEs reside in a translation storage buffer (TSB)
 - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
 - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
 - If match found, the CPU copies the TSB entry into the TLB and translation completes
 - If no match found, kernel interrupted to search the hash table
 - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

We have, we uses TLB to hold translation table entries the for fast hardware lookups. So, this is a cache that makes this access fast, we have got virtual address reference that causes the TLB search. So, if there is a miss, then the hardware will walk into the in memory TSB looking for the TTE corresponding to the address. So that is the standard one.

So, the there is a TLB and if there is a TLB miss then it will be looking into the physical memory. So, if there is no match, then the kernel will be interrupted to search in the hash table and the kernel create will, kernel will then create a TTE, for the appropriate hash table and store it in the TSB. So, that way it works. So, it is quite complex that way.

(Refer Slide Time: 11:02)



Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
 - Each segment can be 4 GB
 - Up to 16 K segments per process
 - Divided into two partitions
 - First partition of up to 8 K segments are private to process (kept in **local descriptor table (LDT)**)
 - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table (GDT)**)

The slide features a yellow background with a blue and orange header. At the bottom, there are two logos on the left and a video inset of a man in a checkered shirt on the right.

Then we have got, this Intel IA 32 bit architecture. So, it supports both segmentation and segmentation with paging. So segmentation with paging means, the segments they are further divided into pages. So the basic problem with segmentation was that, the entire segment has to be loaded contiguously in the memory. So, if you want to avoid that, then we have got a, this segmentation with paging, where each segment is divided into number of pages. So, each segment can be 4 GB long and it use a up to 16 K segment per process.

So, every process can have 16 K number of segments, where each segment is 4 GB at most 4, I can go up to 4 GB. So a huge amount of, logical address space, that is created. So, it is divided into two partitions; first partition is a of up to 8 kilo segments are private to the process and kept in the local descriptor table or LDT and there is second partition up to 8 kilo segments among all processes that is called global descriptor table or GDT. So, there is some local descriptor table and there is some global descriptor table.

(Refer Slide Time: 12:18)

Example: The Intel IA-32 Architecture (Cont.)

- CPU generates logical address
 - Selector given to segmentation unit
 - Which produces linear addresses

s	g	p
13	1	2

- Linear address given to paging unit
 - Which generates physical address in main memory
 - Paging units form equivalent of MMU
 - Pages sizes can be 4 KB or 4 MB

So, this is an example of this IA 32 architecture. So, CPU generates the logical address and this selector given, to the segmentation unit which produces a linear address. So, linear address, so, this selector s this g and p. So, they are this global and a private. So, which this generates the physical address in the main memory and the paging unit is similar to, it is equivalent to actually the MMU the Memory Management Unit page sizes can be 4 kilobyte or 4 megabytes. So, there are two different page sizes for IA 32 4 kilobyte and 4 megabyte.

(Refer Slide Time: 12:55)

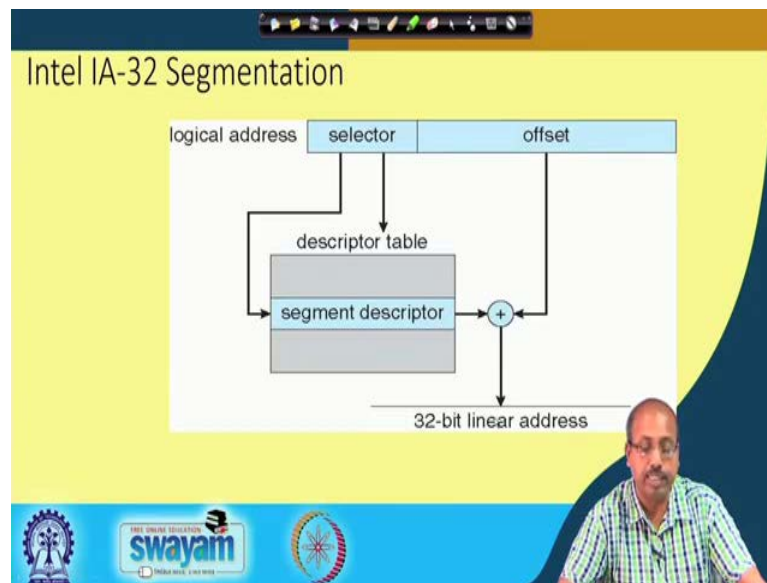
Logical to Physical Address Translation in IA-32

```
graph LR; CPU[CPU] -- logical address --> Seg[segmentation unit]; Seg -- linear address --> Pag[paging unit]; Pag -- physical address --> Mem[physical memory]
```

page number		page offset
p_1	p_2	d
10	10	12

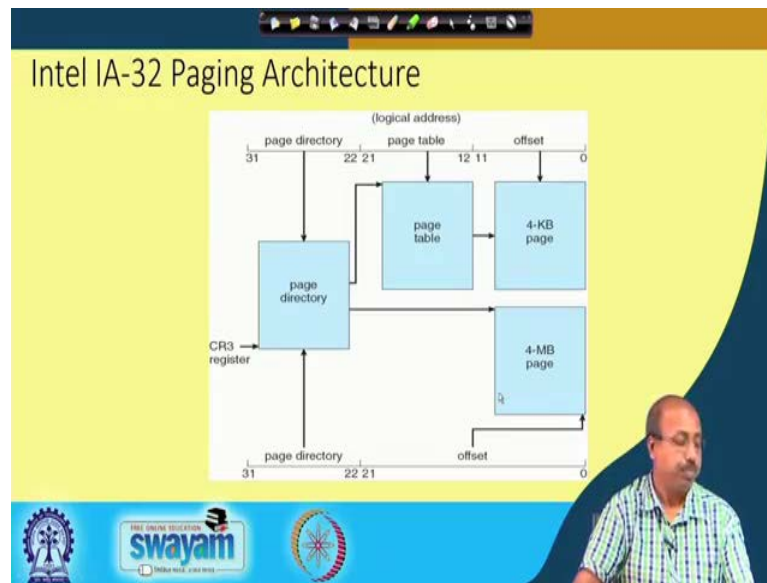
So, the CPU the address, that is generated logical address. It goes in to a segmentation unit and this segmentation unit generates a linear address and this linear address goes to the paging unit and that gives the physical address, that way it is a bit complex. So, page number is divided. So, page number has got two level hierarchy so p 1, p 2. And page p 1 p 2, so, they are the page numbers and then we have got the offset. So, 10 plus 10 plus 12, so that is 32. So, these are the hierarchical pages and this is the offset within the page.

(Refer Slide Time: 13:29)



So, the logical address that we have. So, it has got a selector part that segment is that gives the corresponding segment descriptor and then the offset comes and then the offset may these two are added and then we get the 32 bit linear address from the logical address.

(Refer Slide Time: 13:48)



And then, this logical address that is the 32 bit, so, that is a the page directory part. So, page it is a two level hierarchy as I said for the paging; the first level is called a directory entry and the second entry is second level is the page table. So, directory gives me the corresponding page number from the first 10 bits of this local logical address.

So, we index into this page directory and we get the corresponding page table and from the page table we have got this page table index. So, that will be used to access a, particular page table entries. So, that gives a corresponding frame number and this frame number and the offset. So, the, so, offset is a, 12 bits. So, that is 4 kilobyte page. So, that will be giving us the physical address. So this, a page directory is coming here and this offset is coming here. So, that gives me the final access of the memory location.

(Refer Slide Time: 14:44)

The slide is titled "Intel IA-32 Page Address Extensions". It contains a bulleted list of key points and a diagram illustrating the 3-level paging scheme. The list includes: 32-bit address limits led Intel to create page address extension (PAE), allowing 32-bit apps access to more than 4GB of memory space; Paging went to a 3-level scheme; Top two bits refer to a page directory pointer table; Page-directory and page-table entries moved to 64-bits in size; Net effect is increasing address space to 36 bits – 64GB of physical memory. The diagram shows a 32-bit address split into three parts: a 2-bit page directory pointer (bits 31-30), a 12-bit page directory (bits 29-18), and a 12-bit page table (bits 17-6). The remaining 26 bits (bits 5-0) are the offset. A CR3 register points to the base of the page directory pointer table. The diagram also shows a 4-KB page. The slide features logos for Swamyam and a presenter in the bottom right corner.

- 32-bit address limits led Intel to create **page address extension (PAE)**, allowing 32-bit apps access to more than 4GB of memory space
 - Paging went to a 3-level scheme
 - Top two bits refer to a **page directory pointer table**
 - Page-directory and page-table entries moved to 64-bits in size
 - Net effect is increasing address space to 36 bits – 64GB of physical memory

Diagram illustrating the 3-level paging scheme:

- CR3 register points to the base of the page directory pointer table.
- Page directory pointer table (bits 31-30) points to the base of the page directory.
- Page directory (bits 29-18) points to the base of the page table.
- Page table (bits 17-6) points to the base of the 4-KB page.
- Offset (bits 5-0) is used to access the specific byte within the 4-KB page.

So, there are page address extension. So, 32 bit address limits led intel to create page address extension or PAE allowing 32 bit apps access to more than 4 GB memory space. So, memory space is more than 4 GB. So, how can I have it 32 bit addressing? How can I have such a large memory access?

So, for doing that, so paging it, become a 3 level scheme to a top 2 bits refer to a page directory pointer table. So, this is 32 bits now these 2 bits. So, they are used for a page directory pointer and this page directory pointer and page table entries move to 64 bits in size. So, they are becoming 64 bit.

So, these entries are 64 bit. So, that actually gives this corresponding page directory and this page directory is indexed by this. And then further this, that gives me the page table and page table is indexed by this page table index and then that gives me the final page. So, that is the 4 kilobyte page, but total address space may become 4 GB because these entries that we have. So, these entries may be of more number of bits.

So, page directory and page table entries. So, they are 64 bits in size as a result we can have more address space. So, these individual entries are 64 bits. So, that is how it can have two power the large amount of space compared to power 32.

So, net effect is increasing the address space to 36 bits, to that 36 bits that is 4 GB physical memory. So, it can access 4 GB physical memory by having this page directory

entry. So, this way you can think about this page increasing the number of bits in the page directory and accordingly, we can have more number of, more amount of physical memory though the page size remain same, but the large number of pages can become part of the, part of this of the system.

(Refer Slide Time: 16:46)

Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
 - Outer level has two micro TLBs (one data, one instruction)
 - Inner is single main TLB
 - First inner is checked, on miss others are checked, and on miss page table walk performed by CPU

32 bits

outer page inner page offset

4-KB or 16-KB page

1-MB or 16-MB section

22 bits 10 2 bits

24 bits 10 ⇒ 34

Next we look into the arm architecture. So, this is a, arm is one of a, one of the dominant mobile platform chip, because of it is a low power and all. So this is, a modern energy efficient 32 bit CPU. So, ARM it supports 4 kilobyte and 16 kilobyte pages. So, it supports two types of pages 4 kilobyte and 16 kilobyte and also 1 megabyte and 16 megabyte pages. So, they are called section.

So, it has got two types of thing; one level paging for sections and two level for smaller pages. So, for section it is one level paging for, for smaller pages that is 4 kilo or 16 kilo pages it is two level.

Two levels of TLBs. So, outer level has two micro TLBs, one data and one for instruction and inner one is a single main TLB. First inner is checked on miss the outer will be checked. So, TLB is also divided into two levels. So, inner and outer, first we check the inner TLB, there is a inner TLB miss then it will check in the outer TLB. So, that way it is done.

So, if both of them are a miss in that case then, it will be going through the page table one after the other. So, that is the main memory page table will be accessed and then it will be finding the entry accordingly. So, that is the ARM architecture.

So, in this way you can see that this page table structure, so, this has become very very complex. So, over the newer and faster and faster systems that we want to have and at the same time we want to increase the total amount of virtual address space. So that way if we are the physical memory size. So, that if we are trying to do that then this paging mechanism has become more and more complex and as you are trying to think about a more and more amount of physical memory my CPU is giving me only fixed number of address lines.

So, 32 bit or 64 bit, but with that if we are going to implement or going to put very large amount of physical memory. So, how to do it? So, this has to be done by means of this extension mechanisms in terms of this, this directory entries and all where that, the where that the directory entries can give me more number of bits in the for the address part. So, as I was explaining like a. So, the original address is 32 bit out of that some bits are used to access the directory and after accessing the directory a particular entry in the directory gives me 36 bits.

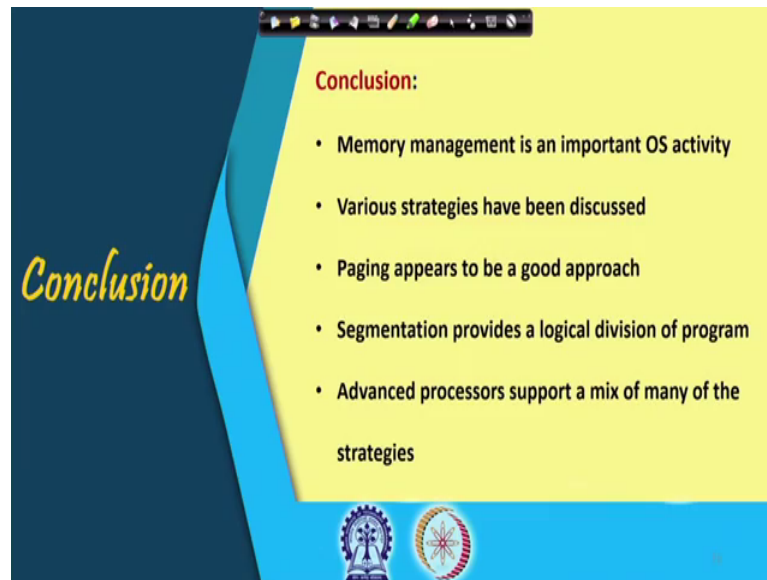
So, as a result now my address becomes a, so, address size is increased. So, that way, so, if it was previously like this that I have got this 32 bit address, if this was 32 bit then some part of it was used for offset, suppose that is 10 k. So, that is the page offset. So, my page size is one kilo only. Now this 22 bit, so, I can have 2^{22} pages only.

Now, if I want to have memory in the range of gigabytes, main memory in the range of gigabytes then this 22 bits are to be they has to go into a directory entry and this from this directory entry. So, this 22 bit will be expanded to more number of bit, suppose I make this directory entries individually 24 bit wide so.

So, the address that I am getting now, is the from the directory entry I have got a 24 bits and we have got 10 bits from here. As a result that overall address that is coming to be 34 bits though the CPU generated logical address of 32 bit. So, now, my physical address has become 34 bit. Now with this 34 bit, so, I can this 34 bit, again I have got division into page and offset. So, offset is the 10 bit and the 24 bit I can use as index into this page table and come to the corresponding physical frame.

So, this way by using this directory based mechanism. So, we can go for higher and higher physical memory interfacing with a processor, that has got less number of bits for the address lines.

(Refer Slide Time: 21:09)



So, with this we come to the conclusion of this discussion. So, this memory management is definitely a very important OS activity and various strategies we have discussed. And we have seen this contiguous allocation; we have seen partitioned allocation, fixed and variable partition allocation, then segmentation then paging.

So, all these schemes, we have seen the paging appears to be a good approach segmentation provides a logical division of the program and we have seen that we have got the segmented paging type of or the paged segmentation type of approach where the segments are again divided into pages. So, that we can combine the benefits of those the both paging and segmentation into one scheme.

So, we have got this advanced processors support a mix of in many of the strategies. So, that is many of the memory management unit. So, if you look into. So, they are going to be very complex and then they are going to give us good strategies for this memory management. So, this hardware support is provided and with that hardware support only, we can go for faster and faster memory interface.

So, with that we come to a conclusion of this discussion on paging. So, we will continue next with the virtual memory policy. So, which one of the very important issues is in today's operating system.