# Computer Architecture

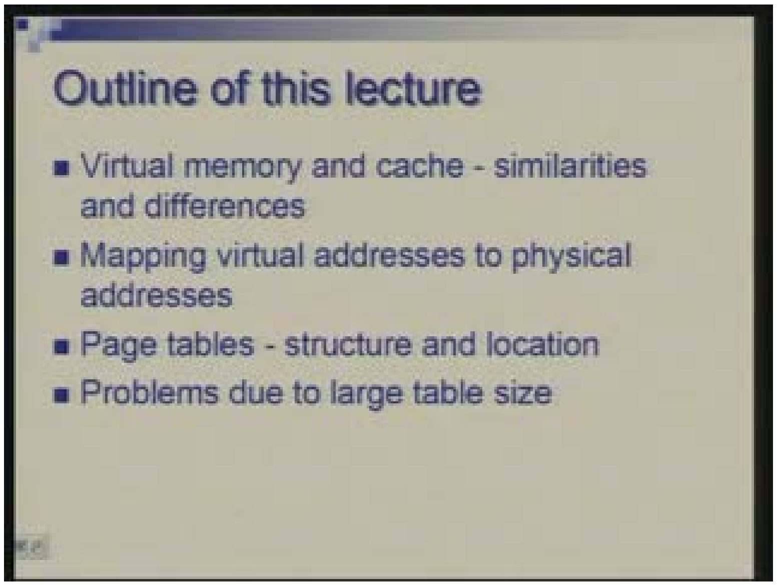
**Prof. Anshul Kumar**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Delhi**

**Lecture - 31**

**Memory Hierarchy: Virtual Memory**

In the cache hierar- in the memory hierarchy, after having discussed the cache organization we move on to next level which is virtual memory. We will first try to compare virtual memory with the cache memory and try to see what are the similarities which we can carry on and where we need to make changes and do things differently.

(Refer Slide Time: 01:11)



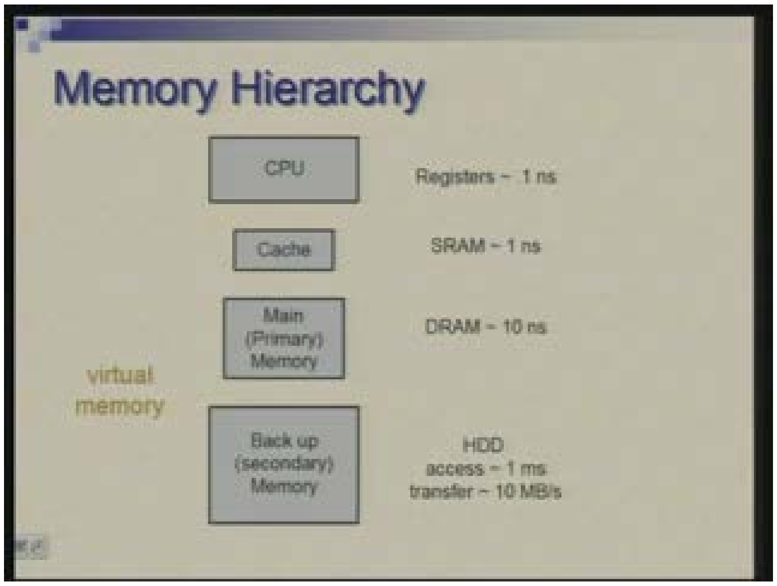
## Outline of this lecture

- Virtual memory and cache - similarities and differences
- Mapping virtual addresses to physical addresses
- Page tables - structure and location
- Problems due to large table size

So we will see essentially this as a problem of mapping virtual addresses to physical addresses so that is how the virtual memory organization is made that you start with virtual addresses and map them to physical addresses. This is done using what is called page table so we will describe what is the structure of page tables, how they operate and where they are located. One major problem which you will have to handle is due to the size of the page tables.

So, coming back to this picture where three levels of memory are shown namely cache, primary memory or main memory and back up memory.

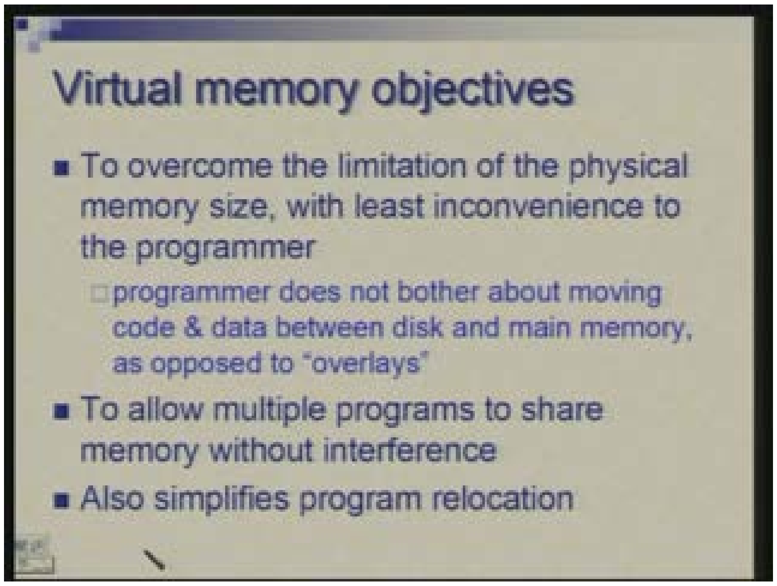
(Refer Slide Time: 02:18)



We have discussed the interface between CPU, cache and main, how CPU tries to access cache and which in turn may update itself from main memory. So, from instruction set architecture point of view, main memory is the one which is typically seen by the programmer and cache is placed in between main memory and processor in a transparent manner in the sense that programmer may not typically be aware of the existence of cache memory. So cache memory is only a device a magic device which is put there to speed up the whole operation. But when you are looking at instructions generally unless there is software driven prefetch you will not come to notice that there is a cache except for performance.

On the other hand, back up memory or the virtual memory organization built around that is used to extend size of main memory and again it is done typically in a manner transparent to the user program. There is the system program which is involved but as far as user program or application program is concerned it may be totally unaware of the fact that there is back up memory which is on a disk. One may only get an impression of a large memory.

(Refer Slide Time: 03:48)



## Virtual memory objectives

- To overcome the limitation of the physical memory size, with least inconvenience to the programmer
  - programmer does not bother about moving code & data between disk and main memory, as opposed to "overlays"
- To allow multiple programs to share memory without interference
- Also simplifies program relocation

The objective of having virtual memory organization is not just to extend the size of the memory but that I will just notice. The main objective of course is to overcome the size limitation of the physical memory and it has to be done in a manner which is most convenient to the programmer, it is as transparent as possible. So, the programmer does not have to bother about moving the data or instruction between virtual memory or the disk and the main memory.

The earlier technique which is more primitive is called overlays where a programmer would explicitly divide the program and data into portions and take care of bringing the right thing in the main memory at right time and also evicting the old contents and the new contents are to be brought in. But virtual memory tries to automate this process, tries to keep in the main memory what is required and keep out what is not required.

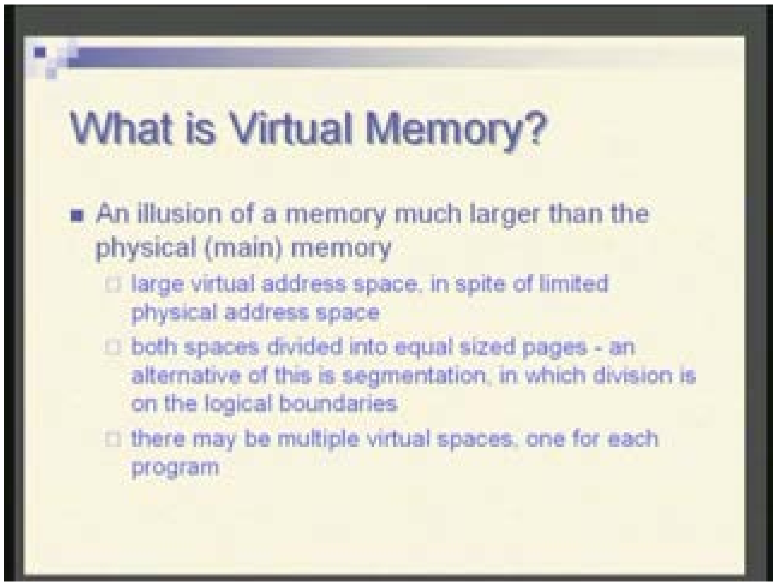
Apart from this main objective another purpose virtual memory organization serves is to allow multiple programs to share same physical memory. Although there may be a single user in a personal typing type of environment but there are many processes which are serving the purpose of the same user. So you have multiple programs or multiple processes which are trying to share the same memory and to do it in a manner which provide protection from one another is also a task combined with the virtual memory organization.

And thirdly it makes it possible to easily reposition or relocate a program in any area of the memory. It could be that, let us say there are two programs A and B on one particular day, A comes in first and gets loaded into earlier part of the memory and B comes later. But on another occasion, B has to be put in the earlier part of the memory and A later or maybe there is a different set of program so you need the flexibility of putting a program anywhere within the memory and that is problem of relocation.

you call recall that If you try to go back to your assembly language programming there are some addresses which are absolute addresses whereas some which are relative. So, for example, in beq instruction you are always going to an instruction in relation to the current instruction. So no matter where the program is placed, if the offset in the beq is let us say 100 you are going 100 bytes ahead or hundred words ahead in that case. So this instruction is relocatable. But if you take a load instruction which takes contents of register and a constant it tries to go to a fixed address and if you position your data elsewhere then this will not..... if you shift your program as well as data somewhere else within the memory then this instruction will have problem.

One device which is often put in assembly language is to have a base register apart from the register which are actually specifying addresses in some local context. So a base register could be set to the datum or the starting point of program plus data whatever the space is allocated and by varying that you can relocate the program. But virtual memory organization, as we will see, also makes the program flexible in terms of where it can be placed or the data where it can be placed. Our focus initially will be on the first aspect namely how do we take care of providing large size.

(Refer Slide Time: 07:45)



## What is Virtual Memory?

- An illusion of a memory much larger than the physical (main) memory
  - large virtual address space, in spite of limited physical address space
  - both spaces divided into equal sized pages - an alternative of this is segmentation, in which division is on the logical boundaries
  - there may be multiple virtual spaces, one for each program

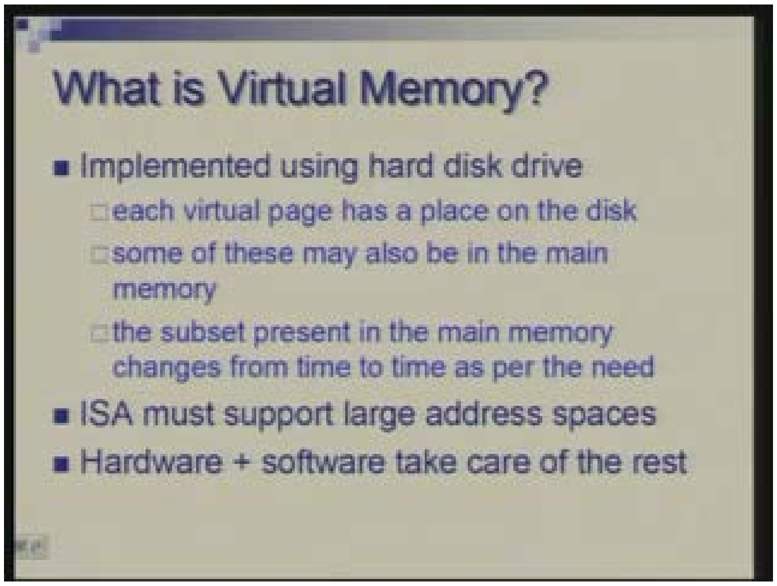
What actually is virtual memory?
Virtual memory is simply an illusion of a memory which is much larger than the physical memory or the main memory. We have basically a large virtual address space which may be bigger than the memory physically present and it may not have any relation with how much memory is present. So you see that machines are bought with 256 MB memory, 512 MB memory or different amounts of memory can be placed within the same processor but a larger physical.... I mean, the total physical space which is available is larger but often it will not be occupied for financial reasons. So you can imagine a larger virtual space where programmer can place the program and data without worrying about the fact that there is physically smaller amount of memory.

The virtual address space and the physical address space, both are divided into sums of equal sizes as we had done with the cache; we had blocks so here we have what is called pages. So virtual memory as well as physical memory, both are divided into pages of equal size and the mapping takes place at the level of page. so now out of entire set of virtual pages which f- which constitute the virtual memory, some pages are placed in the physical memory, some are not so those you need immediately, you need currently are kept in the physical memory and this set could change. You require one such virtual space for each program.

As you remember that I talked about multiple processes, multiple programs trying to share some memory so one could give them each separate virtual space of very large size. Now here we have talked of dividing both the spaces into areas of equal size. There is another possibility of doing it differently in the sense that you divide more on a logical plane. that is, for example, if you have functions, you could say that each function or may be group of functions or each data structure or group of data structures could be logically organized as what you may call as segment.

So the whole program plus data is divided into a few segments which may not necessarily of same size and then you can talk of keeping some segments in the physical memory some out of it. The advantage of this is that you have what you have is a complete logical entity. so as long as long as you are executing a function you have the entire function in the memory or while you are working with one data structure the whole thing is there in the memory whereas page is something which is artificial. You are taking a program and chopping it off into equal parts which makes things convenient. So organizationally dividing into pages is very convenient and it is very efficient.

(Refer Slide Time: 11:18)



## What is Virtual Memory?

- Implemented using hard disk drive
  - each virtual page has a place on the disk
  - some of these may also be in the main memory
  - the subset present in the main memory changes from time to time as per the need
- ISA must support large address spaces
- Hardware + software take care of the rest

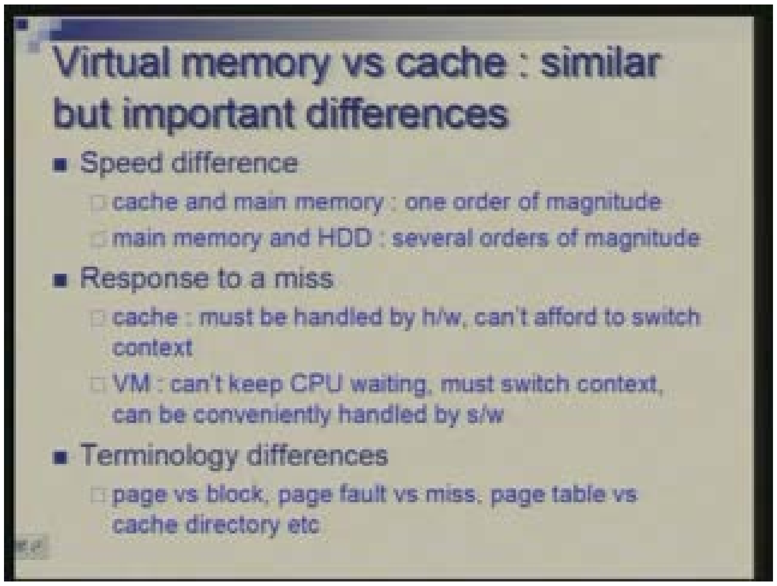
So how do you implement virtual memory?

You implement basically by relying on hard disk drive which is firstly a non-volatile medium and secondly it has larger capacity at a lower cost. So we begin by assuming that all virtual pages are primarily placed in hard disk. In some area you can place them and keep some of them in the main memory and it is this subset which is in the main memory can be made to change overtime as the need arises.

So now, for all this to happen, the instruction set architecture should support a larger disk space. So in MIPS architecture, we have discussed for example, there is an address space of 32 bits, addresses of 32 bits which means 4 gigabyte of space is there. So we can say that a programmer can always imagine a 4 gigabytes of virtual space and might work with smaller amount of memory may be a few megabytes.

So who takes care of the risk?
It is some hardware support which is there in the processor which the programmer may not directly see plus software which is basically the operating system software or the system software. So now for doing all this can we exactly like what we do for the cache?
The answer is yes. There are similarities. The whole idea is basically the same that from lower level of hierarchy you keep some information at the higher level of hierarchy and change it as and when necessary. So, as far as that is concerned things are similar. But there are some important differences which need to be borne in mind while organizing the whole thing.

(Refer Slide Time: 12:44)



## Virtual memory vs cache : similar but important differences

- Speed difference
  - cache and main memory : one order of magnitude
  - main memory and HDD : several orders of magnitude
- Response to a miss
  - cache : must be handled by h/w, can't afford to switch context
  - VM : can't keep CPU waiting, must switch context, can be conveniently handled by s/w
- Terminology differences
  - page vs block, page fault vs miss, page table vs cache directory etc

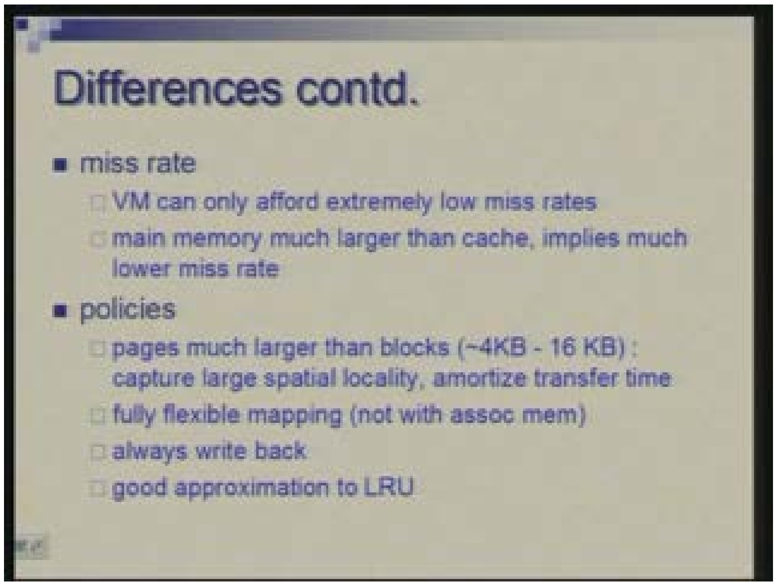
The main difference essentially comes from the fact that speeds are different. So when you talk of speed difference between cache and main memory we notice that it is only about an order of magnitude difference. Let me quickly project this figure (Refer Slide Time: 13:39). So the difference between these two is much smaller; difference between these two is very very large. So the techniques which worked here may not always work here in the same manner, we need to keep this fact in mind that HDD is much much slower than the main memory, several orders of magnitude.

So now the response to a miss when you do not find things in the cache what do you do? You basically you expect that a few more cycles are required so you just hold the processor and do the needful and continue execution of the program. So this action has to be handled by the hardware, you cannot have a special software doing this because that will require many many cycles. So hardware can quickly get a block from main memory and serve a miss. We cannot afford to switch context because context switch means that instead of waiting, you do something else so that is not possible because that changeover may require large amount of time.

On the other hand, when you are working with virtual memory you find something which you are looking for is not there in the main memory, you need to go to disk the time involved is very very large now, milliseconds, so you cannot keep CPU waiting for that long and you must switch the context. If you have several milliseconds of gap the processor may better do something else. So from one process or one task it switches to something else and the response to this miss can now be handled by software. Because we have time available so we can do it more conveniently by software.

Apart from these differences which are coming because of different speeds the terminology is also different for more for historical reasons, things have come up differently. we are talking of pages instead of blocks, we are talking of we talk of page fault instead of miss and as you will see later there is a page table instead of a cache directory so that is a switch in terminology difference say some tens to hundreds of thousands then miss rate has to be accordingly very very small otherwise you have a very large figure at hand and you will lose tremendously in terms of performance.

(Refer Slide Time: 16:02)



## Differences contd.

- miss rate
  - VM can only afford extremely low miss rates
  - main memory much larger than cache, implies much lower miss rate
- policies
  - pages much larger than blocks (~4KB - 16 KB) : capture large spatial locality, amortize transfer time
  - fully flexible mapping (not with assoc mem)
  - always write back
  - good approximation to LRU

Now this is..... miss rate is indeed small here, 1 because of large physical memory size as compared to small cache size. The miss rate which you see in context of cache is largely determined by how big the cache is, larger the cache smaller the miss rate. So basically it depends upon how much of space you are capturing in this level of memory, if this is large miss rate is lower whereas cache could be in terms kilobytes, main memory is in terms of megabytes so naturally in the first instant itself you have a much lower miss rate which is good. But apart from that we have to organize other things also which helps in keeping the miss rate as low as possible.

So one is that page is to be kept much larger than what the block size is as we have seen. In cache we cannot increase the block size too much because it will increase the miss penalty and it will also reduce the number of blocks and therefore the localities you are capturing will be small so you cannot have two larger blocks, you have basically a few words four words, sixteen words utmost sixty four words but generally not larger than that. Here we need to have large page size firstly so that you can capture much larger locality.

Secondly, when you are getting data from disk it does not make sense to just a few words because once you spend few milliseconds to reach a particular position in the disk you better transfer substantial amount of data so that the time to access is amortized over larger number of words. So, for both these reasons you have typically a page size which is 4 kilobytes to 16 kilobytes but the trend is to increase it even further to 32 or 64 kilobytes.

Secondly, the mapping in case of cache we have seen that there is direct mapping, associative mapping and set associative. The most common is set associative with degree of 2, 4, 8 or something. But here we need to do our utmost best that is we go for fully associative mapping. Although we do not use associative memory for that but we need complete flexibility in terms of mapping so that there is no miss because of the conflicts. How we do that we will see later.

And among the choice of write back and write through we need to use write back, write through does not make sense because it does not make sense to write one word into disk so you are going to write always a page which means that you have write back. Write back choice also reduces the number of misses. The negative point of write back it caches that the time to write back is larger, time to write a word is smaller but the difference is that if you are writing a word in write through cache you are writing more often, if you are doing write back you are writing less often. So what suits here is a write back approach and you do not have any virtual memory with write through it is always write back.

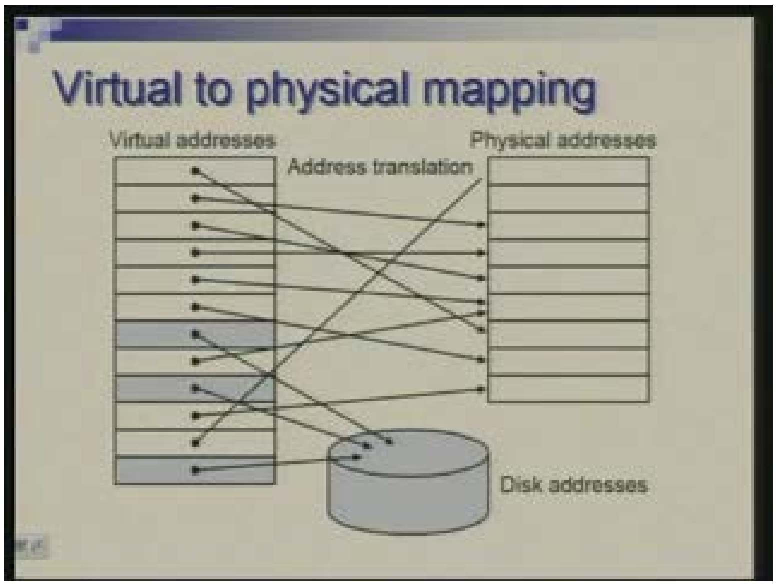
Then the last point is what replacement policy you use. This is also important because if you are throwing back throwing away a wrong piece of information then you are losing, you will have a miss later on. The ideal thing as for your replacement policy would be to be able to see in future. So, if you know what references are going to occur in near future it is that data or instruction which you need to retain. Unfortunately that is not realistic.

What we can is we can only look at the past and then try to figure out what we are likely to use. So on that basis LRU has been experimentally found to be the most appropriate policy but it is not easy to implement. In concept it sounds similar that you pick up the one which has been least recently used but how do you implement.

If you are tagging each block or in this case each page with time then this is something this is a quantity which is not bound, unbounded quantity so it is not practical to use time or the cycle number because you do not want to restrict programs to run for a limited time, you have people doing research who put their programs for execution for days together and then get the result. So that is not really workable.

You might think of, may be maintaining an order which was most recently used, keep it number call it number 1 which was next recently used call it number 2 and so on then you shuffle the order as accesses take place. Again the problem may be that you may have to make changes in the various..... suppose you are maintaining order of various pages or various cache blocks and you change the order of one so the others have to be shuffled or you have to use more sophisticate data structures to do that which requires that to make one access to one level of memory you to do this housekeeping of LRU you may have to make many accesses and the purpose may get lost trying to do this LRU nicely. So, doing it in hardware is in fact more difficult and doing it in software somewhat is easy but still we do not want to lose efficiency. in hardware that means what I mean is when you are handling cache you may you may not worry about doing something which is close to LRU, you may take a more approximate policy but in case of virtual memory you try to be as close to LRU as possible.

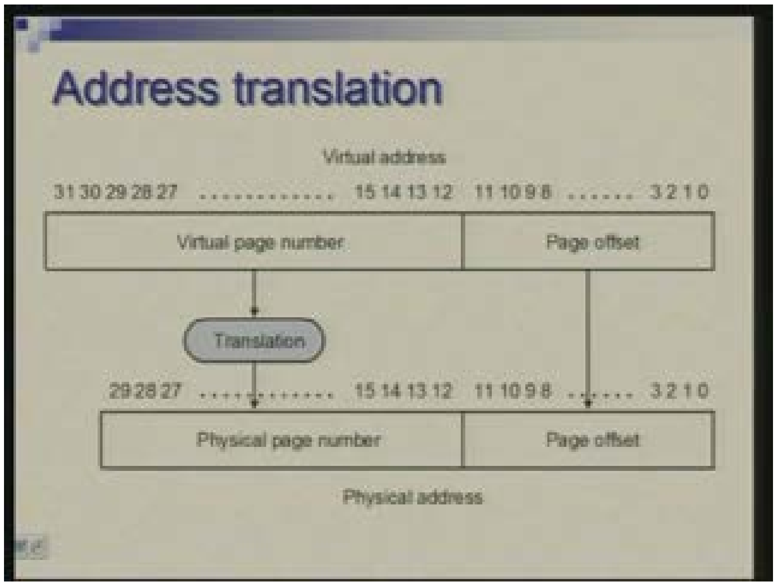
(Refer Slide Time: 23:45)



## Virtual to physical mapping

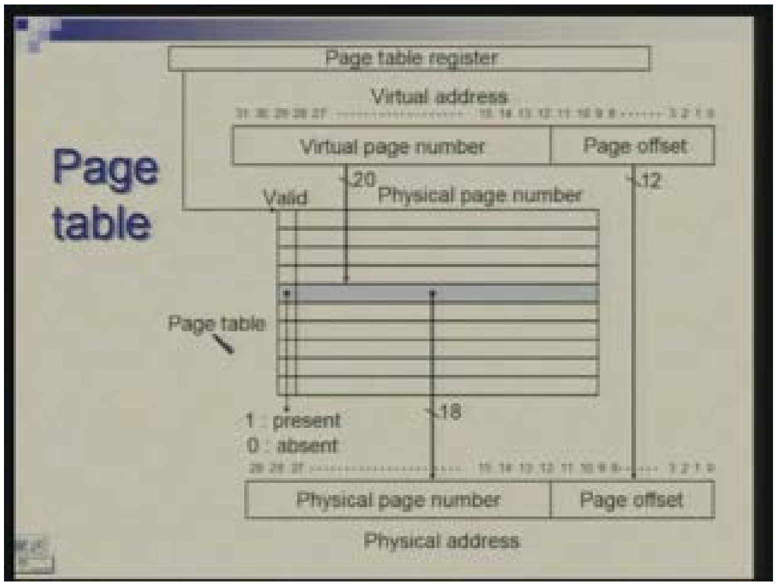
So now with this background let us try to look at this picture which shows virtual memory on one side and physical memory on the other side. Each is divided into pages, each page of virtual memory each virtual page is either mapped to physical memory or to a disk. In fact strictly speaking each page must have its residence in on disk; some of them will also have residence on the physical memory. So you need a mechanism which defines what goes where; given a page where we place it in the physical memory and once we have placed it how do we find it later on. This process is called translation of the address

(Refer Slide Time: 24:34)



Imagine that you have virtual address; I am showing 32 bits in this case, this could be divided into page offset and virtual page number. So suppose page size is 4 kilobytes which mean a 12-bit number will specify byte within a page, rest of the address is the page number. We have this 20 bits specifying page number and 12 bits specifying address within a page. Suppose we have physical memory present which is 2 raised to the power 30 bytes so let us say this is 4 gigabytes and this is 1 gigabyte (Refer Slide Time: 25:24) now physical memory can physical memory address can be similarly divided into page number and offset so the problem now remains is to translate virtual page number into physical page number so it is a translation process we need to figure out how it has to happen.

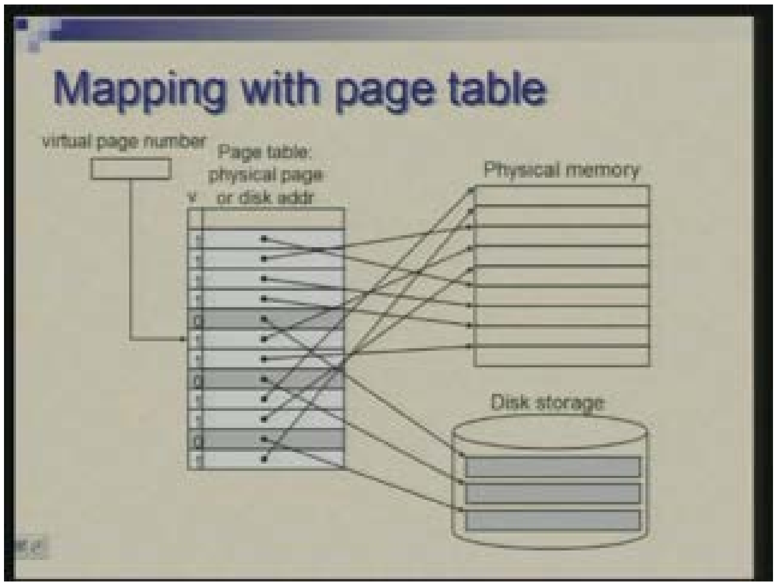
(Refer Slide Time: 25:48)



This is the mechanism using page table. So page table is basically a lookup table where you keep one entry for every virtual page. So, given the virtual page number you look at appropriate entry here and this will tell you where this page is located in the physical memory or effectively it will give you physical page number. There is a bit apart from this physical page number which is a valid bit and has similar purpose as we have seen in cache, it will tell that whether this particular virtual page is present in physical memory or it is not present. So, if it is not present in the physical memory we need to know address of this page in the disk because we have to get it from the disk. So either we get physical page number here or we get the disk address or pointed to area where disk address would be stored and the location of this table could be obtained from a register which is called page table register. So it is a very simple lookup process.

Now compare it with cache, how we were doing. In cache we did not in direct map cache we looked at the index bits and directly went to cache where we simply make a check whether it is present or not. We compared with the tag bits. But here now we are before we access the higher level of memory which is the physical memory in this case we are going through this table to reach the position whereas in case of cache we directly reached the cache memory through that index but that is the direct memory access where the locations are fixed. In terms of its effect this is same as fully associative cache where in case of cache we were required to make comparison with all the tags present for various blocks. But in this case we are not making any comparison. Here you have one entry per virtual page. In cache it is one entry per cache block. So the number of entries in cache directory is equal to the number of blocks in higher level memory. Here it is the number of entry equal to number of entries in the number of pages in the low level memory so that is the difference you should notice.

(Refer Slide Time: 28:42)

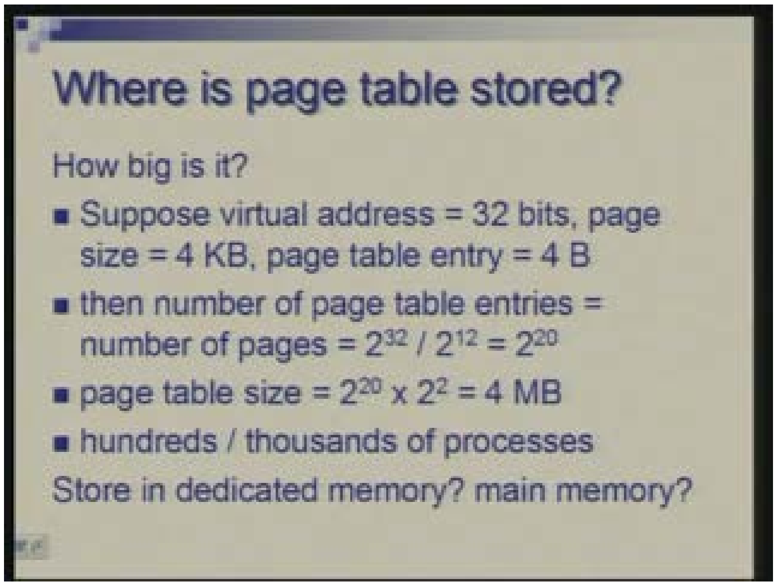


## Mapping with page table

This picture shows the mapping process with looking at the virtual page number. taking it Taking it out of the virtual address we are indexing into this table and this table (Refer Slide Time: 28:57) is telling, it is allowing us to either go to physical memory or to disk as the case is. So, if you have a hit it is fine you make an access, if you have a miss then this is called page fault and page fault result in context switch. So first of all the current process which had made this request is suspended. you initiate a request to the disk to do the transfer, meanwhile the disk is ready several milliseconds are going to elapse and you can execute thousands and thousands of instructions so control is transferred to another process which is waiting for execution.

So now the question is where is the page table stored. What we have said is before you go to the physical memory you have to go through page table, you have to make an access but where is that located. And the answer came we worked out if we understand how big that table is or what is the space requirement .

(Refer Slide Time: 30:02)



## Where is page table stored?

How big is it?

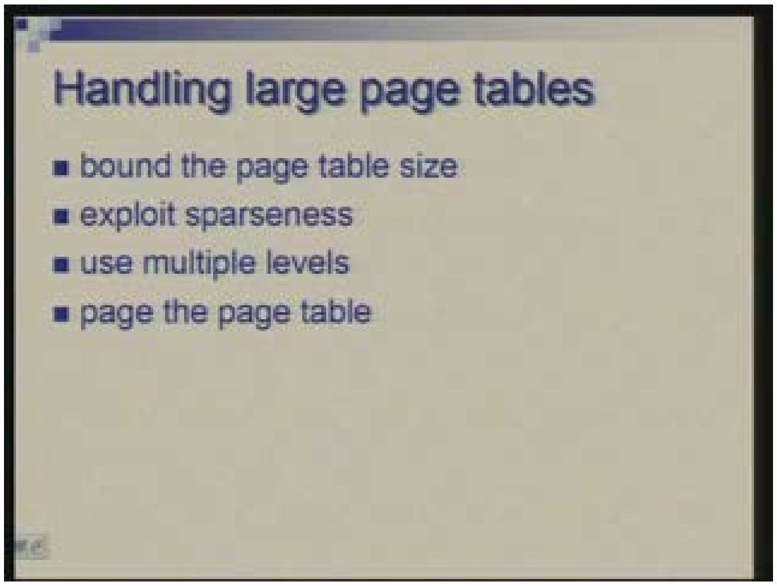
- Suppose virtual address = 32 bits, page size = 4 KB, page table entry = 4 B
- then number of page table entries = number of pages = $2^{32} / 2^{12} = 2^{20}$
- page table size = $2^{20} \times 2^{2}$ = 4 MB
- hundreds / thousands of processes

Store in dedicated memory? main memory?

Let us take some typical example. Suppose virtual address is 32 bits, page size is 4 kilobytes and each page table entry is 4 bytes. Now what I indicated was that page table entry as a valid bit and physical page number. Actually apart from this it may have other information, it may have information about memory protection in context of multiple processes. So let us imagine that you have 4 bytes of information per entry, you put all these together the number of page number of page table entries which are equal to number of pages is 2 raised to the power 32 virtual memory size divided by page size or 2 raised to the power 20. So 1 million pages are there in the virtual memory and size of the table could be obtained by multiplying this 1 million by size of each entry and you get 4 megabytes.

Now there are hundreds and thousands of processes each one has to worry about and we have decided to allocate same amount of virtual space to each of these. Then where are we going to put this much of information. Having a direct separate memory for this is ruled out so your attention goes to main memory. But main memory is also in terms of megabytes few hundreds of megabytes at its best. So, trying to stuff it with several hundreds of page table will simply leave no space for other useful things. So what do we do?
There are various ways you can handle it. Some of these are listed here, common ones that you realize bound on the page table size, exploit that or exploit the sparseness that means the whole page table may not be really active, you can use multiple level or you can use techniques like paging the page table and so on. So let us look at these one by one.

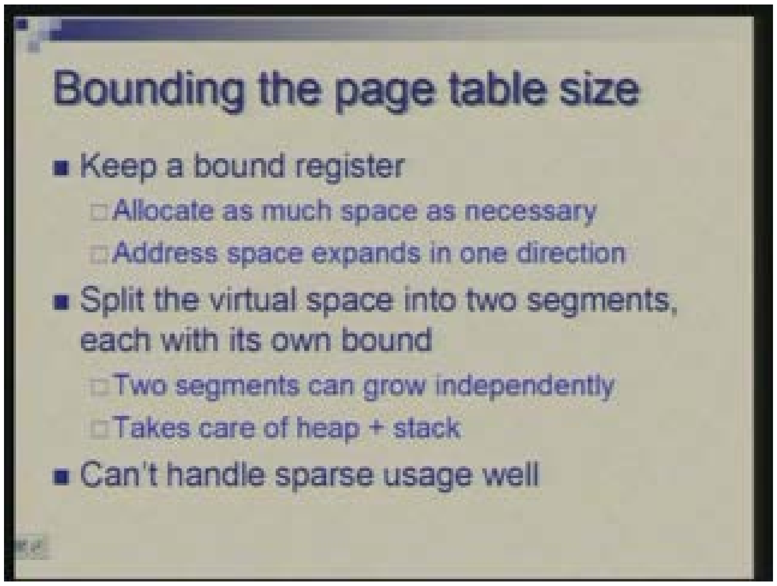
(Refer Slide Time: 32:15)



## Handling large page tables

- bound the page table size
- exploit sparseness
- use multiple levels
- page the page table

(Refer Slide Time: 32:18)



## Bounding the page table size

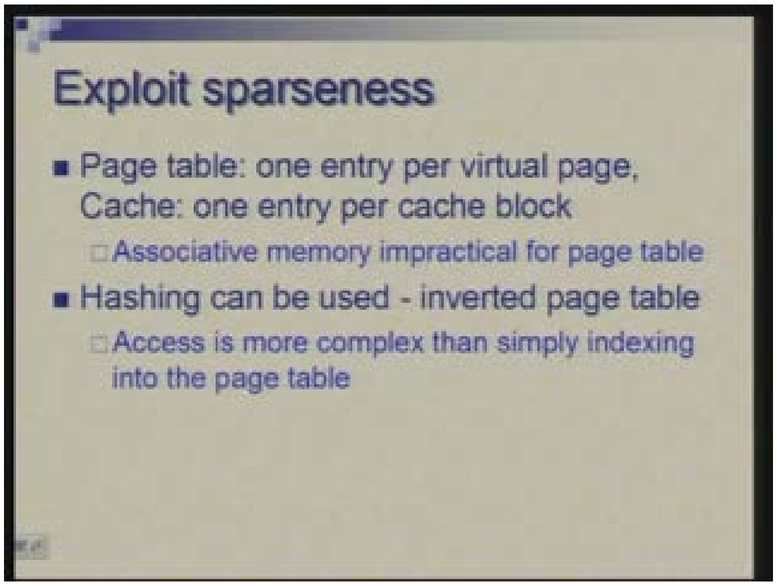
- Keep a bound register
  - Allocate as much space as necessary
  - Address space expands in one direction
- Split the virtual space into two segments, each with its own bound
  - Two segments can grow independently
  - Takes care of heap + stack
- Can't handle sparse usage well

So, although we are saying that you give 4GB space 4GB of virtual space to each program but it may not need that. Suppose it needs only a few hundreds of megabytes or may be few tens of megabytes then we should actually cater for only that so we can have a register which keeps track of the bound and once you reduce the memory virtual memory size the page table size also reduces. So allocate only as much space as necessary and allow it to expand or grow in one direction if need of a program changes dynamically. But this is not really does not really match with the requirement.

Typically the programs are organized to grow in two directions: one growing area is stack which grows or shrinks as calls are made to functions. On the other hand, there is also what is called heap which grows and shrinks as memory is randomly allocated and deallocated through through the pointers. So typically these are organized to grow in two opposite directions. Suppose you keep some space stack grows in stack grows in one direction, heap grows in another direction so that you can allow them to grow independently. So it is not difficult to accommodate this by, thinking of this as two virtual memory segments and have two page tables or you can think of two parts of a page table which can be made to grow independently so this can take care of both stack and heap. But this is not sufficient. It does not still reduce the memory requirement substantially.

One thing we are not able to do here is that we are not able to handle sparcity of the table because in the table we are keeping one entry for every virtual page whereas the pages actually present in the physical memory are much fewer so why not we just keep track of those. If you have a mechanism to just keep track of only those entries of page table for which we have current requirement then it will be much better.

(Refer Slide Time: 34:52)

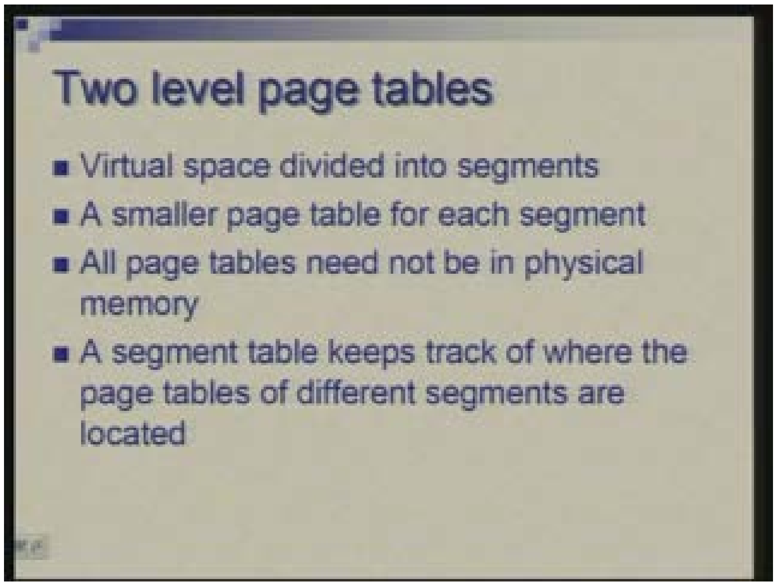


## Exploit sparseness

- Page table: one entry per virtual page, Cache: one entry per cache block
  - Associative memory impractical for page table
- Hashing can be used - inverted page table
  - Access is more complex than simply indexing into the page table

One possibility is to go back to cache like approach that is actually to have associative memory where the number of entries would correspond to number of entries actually there in the physical memory so that will be much smaller. But even then having associative memory of that size is impractical. So alternative is to use hashing technique. Given a virtual page number, you apply suitable hashing function, you go to an you get an index which takes you into either a smaller table or takes you or gives you directly the physical page number. This is what is called inverted page table because you will have entries corresponding to I mean it is becoming something like cache situation where you will keep entries organized as per the entries which are there in the physical memory.

(Refer Slide Time: 36:11)

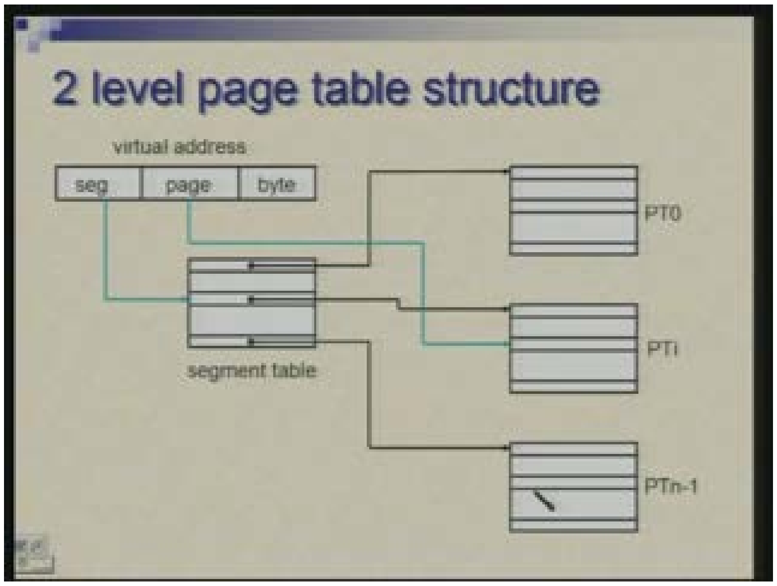


## Two level page tables

- Virtual space divided into segments
- A smaller page table for each segment
- All page tables need not be in physical memory
- A segment table keeps track of where the page tables of different segments are located

A more common technique is to use either two level page table or paging the page table as I will see little later. So what we are doing here is that we organize the virtual space in terms of segments and segments are divided into pages. So now it is different from segments which I mentioned earlier which correspond to the logical boundaries or function in data structure. So let us imagine that we have segments of equal size. It is just that entire virtual memory is divided into some larger chunks which are again divided into smaller chunks which are called pages. So we can have one page table for each segment then you have flexibility of not having to keep all the page tables in the main memory, you can keep a few page table which are relevant in the main memory and therefore reduce your space requirement. The segment table will keep track of where the page tables are, which of them are in main memory which are there in the disk so this organization will try to just track information which is active and which are required.

(Refer Slide Time: 37:26)



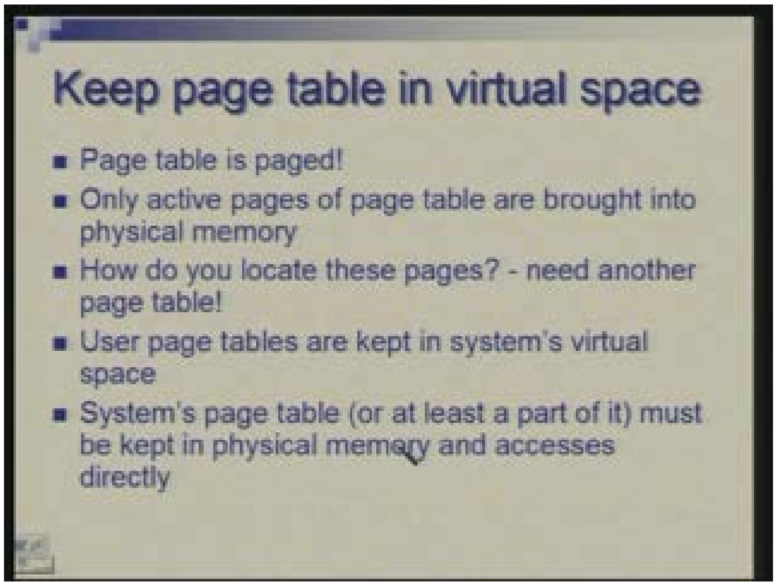
## 2 level page table structure

So conceptually this is how it is organized. You have a number of page table, page table 0 1 2 3 4, page table i, page table n minus 1 these are much smaller page tables. Effectively if you put all these together they will form your good old single monolithic page table. But now we have divided and all of them need not reside in the main memory. The starting addresses of these page tables is pointed out by another table which we are calling segment table. So first you make an access to segment table. Now imagine that virtual addresses divided into three parts: segment number, page number within a segment and byte number within a page. So using the segment number you pick up one entry in the segment table this will tell you where the page table is.

Let us imagine that page table you are looking for is in the physical memory. So this will give you the starting address of that page table. Within this page table you can take this page number and index into it (Refer Slide Time: 38:27). This will now tell you the physical address or physical page number of the entry you are looking at you are looking for and therefore by making this two-step access you can make you can reach the required point in the physical memory without necessarily having to have this whole thing in the physical memory.

Now it could be that the page table you are looking for is not in the physical memory so a page fault will occur at this point and first you will bring this page table into physical memory then make an access to the page table and then make access to the main memory. The starting address of the segment table could be in a register. As you can see here there are two points, two places where you may encounter page fault. You may encounter page fault when accessing a page table itself; secondly, you may access you may have the page table but you may not have the page so page fault can be encountered at any of those two levels.

(Refer Slide Time: 40:00)



## Keep page table in virtual space

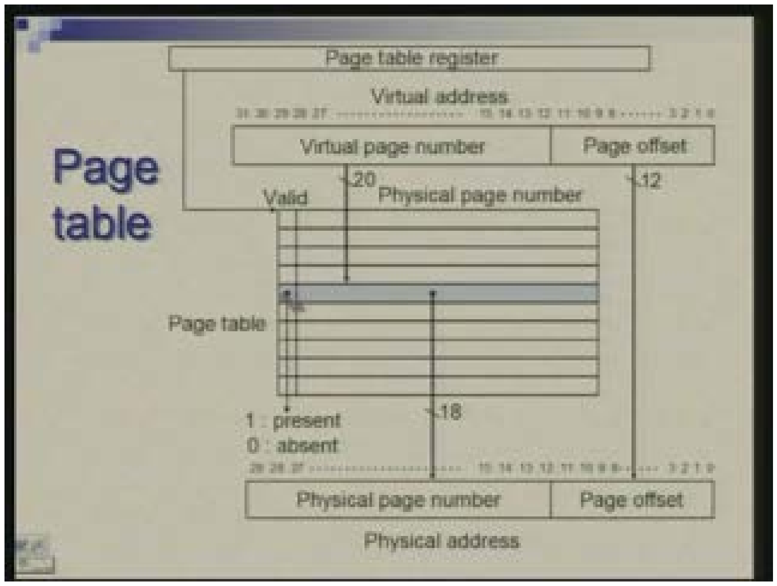
- Page table is paged!
- Only active pages of page table are brought into physical memory
- How do you locate these pages? - need another page table!
- User page tables are kept in system's virtual space
- System's page table (or at least a part of it) must be kept in physical memory and accesses directly

Lastly, we can think of keeping the page table itself in the virtual memory. Instead of imagining page table to be in the physical memory and worrying about two levels or other techniques suppose you place the page table itself in virtual memory then automatically some part of it will be kept in physical memory, some will not be rest will not be. so it is only you will bring only that part of the page table as it is needed, as you make accesses few entries will be there rest will not be there.

Now how do you locate this page table?
As you have seen that to access any memory location you need to go through page table.
If page table itself in virtual memory you need some way to find out where it is in the physical memory. Because if you are keeping page table in the virtual memory how do we get it, so it is a kind of vicious cycle. What is done is that we can keep user page tables in system's virtual space. You have many user processes, you have a system process. So system is also assigned a virtual space. Let us keep user page table in system virtual space and we can also ensure that system space table or at least some part of it is always there in the physical memory so you can start by that, find out where your page table is or you are not worried about having the entire page table in the main memory, you want to find that part of the page table where you are making reference. So the address of entry you are looking for is given by the starting address of the page table plus the offset within the table.

Let us get back to this.

(Refer Slide Time: 42:02)



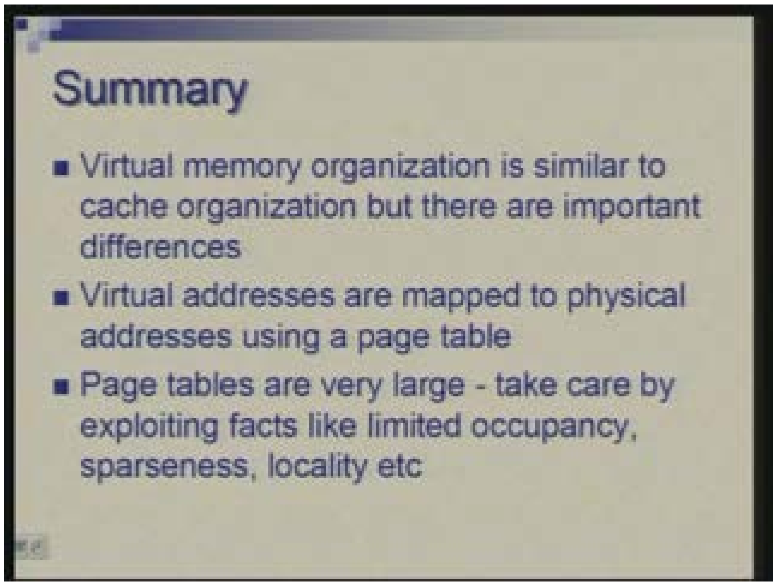
How are you making an access to a particular entry in the page table?
You need the starting address of the table and you need this offset which is given by virtual page number. So if let us say page number if 10 that means you want tenth entry so in terms of bytes it may be 10 into 4. So you take this address plus four times this number that gives an address; this is the address of...... normally imagine if page table was in the physical memory, you will take this address and go to physical memory directly, read the page table entry and proceed further.

But now what we are saying is that this address which you have it is an address of one page table entry this itself is a virtual address. So first we will have to get a physical counterpart by going through system's page table because this is in system virtual space so we go through system page table which for the moment you have to imagine is in the main memory so you access that, get the physical address of the page table entry hoping that it is present in the main memory you make an access there, find out whether page you are requiring ultimately is present or not and then make an access so you need to go through these two-step process.

The first step is accessing the system's page table which hopefully is in the physical memory, then accessing user's page table which the relevant part hopefully may be in the physical memory otherwise there will be a page fault, once you got that then you make access to physical memory. So it is basically not two, totally three steps are involved to read anything from virtual memory.

I will elaborate these little further pictorially so you will have a clear idea and then we will also go to performance issues next time.

(Refer Slide Time: 44:13)



## Summary

- Virtual memory organization is similar to cache organization but there are important differences
- Virtual addresses are mapped to physical addresses using a page table
- Page tables are very large - take care by exploiting facts like limited occupancy, sparseness, locality etc

So let me summarize at this point. We started by virtual memory and cache organization and noticed similarities, also noticed the differences. Differences are important otherwise things will not work with the efficiency we want. The key mechanism here is the page table which is used to translate virtual addresses to physical addresses. Page tables are tend to be very large which poses a problem and there are many techniques to counter that. These techniques basically exploit the limited size, the sparse sparseness, locality and so on and ultimately the relevant part of page table has to be in the physical memory; there is no separate memory kept for this. I will stop at this point, thank you.