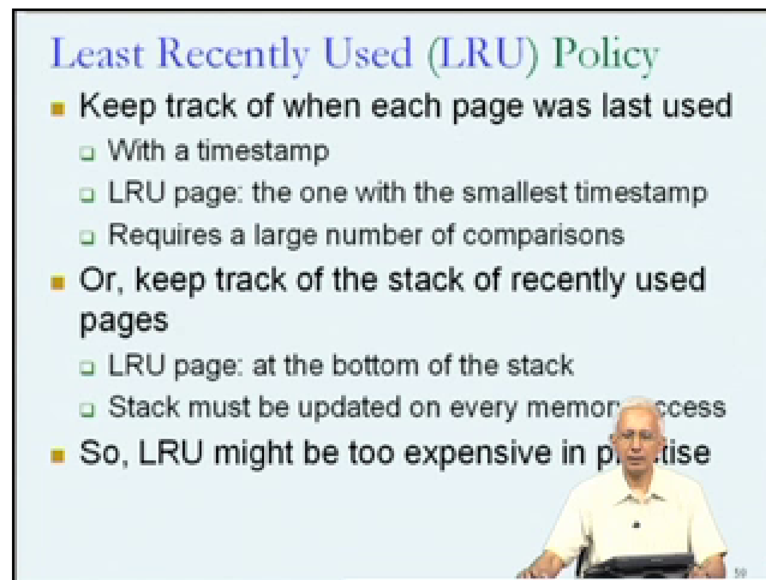
High Performance Computing

Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture No. # 15

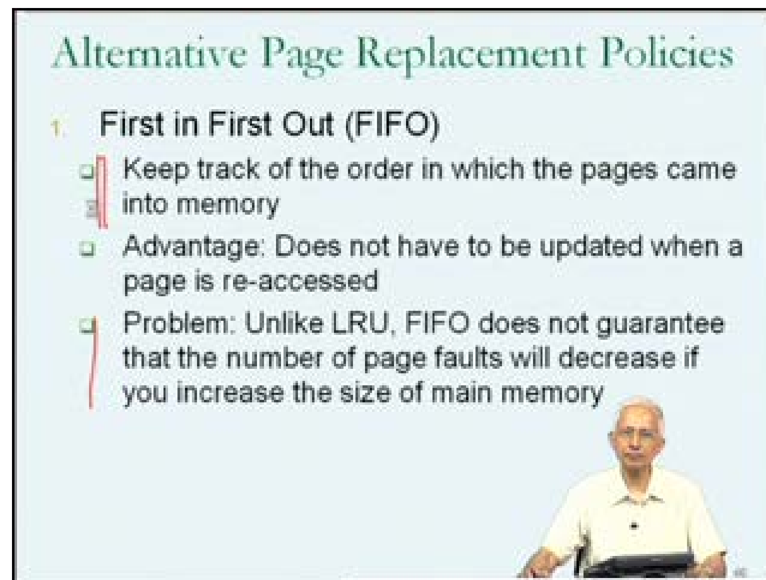
Welcome to lecture 15 of the course on, High Performance Computing. We had entered lecture 14 by understanding a principle, a basic principle of program behavior, which might be useful in part of the operating system. This was the Principle of Locality of Reference. We saw how this principle might be used, inside the operating system page fault handler, to make decisions about page replacement, and had come up with the idea the least recently used page replacement policy.

(Refer Slide Time: 00:40)



We looked at two possible ways that the page LRU page replacement policy could be implemented and unfortunately came to the conclusion that it might be too expensive to use in practice.

(Refer Slide Time: 01:01)



So, at the end of lecture 14, we had suggested that there must be alternatives; alternative page replacement policies that operating systems might use. Inside the operating system, the decisions must be made using policies they can be that are feasible. They can be executed using very small time over heads.

The policy itself, the decision making should not swamp the execution time of your program. So, one of the possible ideas, which would come to mind is what I will call first in first out, and the idea of first in first out is that among all the pages, which are currently in main memory, when the time comes to make a page replacement decision, we could choose to remove from main memory that page, which has been in main memory the longest time. In other words, we keep track of the order in which the pages came into main memory, not the order in which they were referenced.

Now, this is the problem with the LRU page replacement policy. The reason that the heavy over heads and it is infeasibility came up was because I will be either have to do things with time stamps, huge numbers of comparisons of time stamps or we have to do operations which were expensive in terms of the need to update a data structure such as the time stamp itself or the LRU stack on every memory access, whether or not it was a page fault, and this interrupt being a very expensive future of a page replacement policy as far as its implementation in operating system code.

Now, in first in first out, all that we have to do is to keep track of the order in which the pages come into memory and pages come into memory only in the event of page faults. And when the page just come into memory, basically this means that we just not have to keep track of the time at which the page came into memory and therefore, the over heads are substantially less than the over heads of the LRU like page replacement policies. To identify the page which has been in memory the longest, we just have to keep track of it specially and therefore, the operating system does not have high over heads.

There is no need for large number of comparisons; there is no need to update any data structure on every memory access. Therefore, this sounds like a very good idea. It is also appearing from the perspective of taking into account, at least something to do with the point in time at which, a page became of interest to the program, in other words, something relating to the past history of the program. Now, as I said, the advantage of the scheme is said it does not have to be updated when a page is re-accessed, and therefore, the over heads of implementing this idea, first in first out, would be substantially lower than the LRU that we saw.

The disadvantage is that the FIFO policy suffers from subtle, but very tricky problem, which is that unlike the LRU least recently, used policy, it does not actually guarantee that the number of page faults that occur when your program executes, will decrease, if you run the program after increasing the amount of main memory. In other words, if you double the amount of main memory that your program has and then you run the program again, there is no guarantee that the number of page faults would come down. That is apparently what the statement is making or what the statement is claiming.

Now, this property of FIFO is well known, it is known as, Beladis Anomaly, and in fact, has come in the way of FIFO, being widely adopted as a page replacement policy, but from our perspective, if you think about the principle of locality of reference and you think about the object with the hypothesis behind FIFO, then you notice that FIFO is assuming that a page which has been in memory for a long time, it is unlikely to be referenced in the near future, whereas, the page which has only recently come into memory, is likely to be reference in the near future.

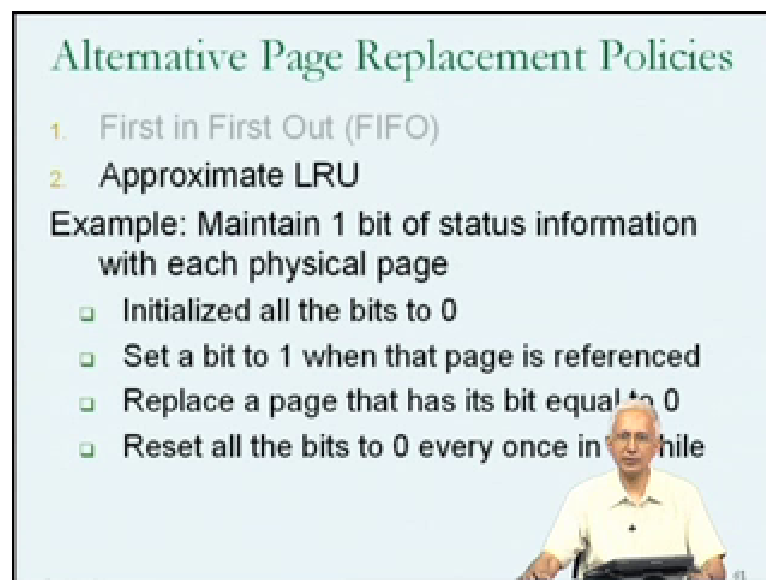
Now, the second statement may be in sync with the principle of locality of reference. In other words, the page which has come into memory recently is likely to be referenced in

the near future. But the first statement is not. The page which was referenced a long time, the page which was bought into memory in the distant past may have been referenced repeatedly, and frequently and recently by my program.

Therefore, if I use the point in time timestamp of when the page came into memory for replacement decision, there is every chance that the page that is selected for replacement will be required in the very near future. It may, in fact, be the page that was referenced very recently.

So, FIFO does not actually take into account the consequences of the principle of locality of reference and is therefore, not to be considered as a good policy for page replacement. It is unlikely that you will find operating system using FIFO for making the page replacement decision.

(Refer Slide Time: 05:41)



Now, what might be more reasonable is since LRU is in sync with the principle of locality of reference, which we except as a believable program property, and for which we can write programs that do show that property. Then rather than trying to implement LRU exactly, we could try to come up with approximate implementations of LRU. In other words, page replacement policies, which are in this period of LRU, and this is an interesting idea. Let me give you one example of an approximate LRU, an attempt to mimic LRU. Now, the intent of this particular example is to minimize the amount of

information that is maintained and the frequency with which information has to be updated, the frequency does and the number of comparisons that have to be made.

So, frequency, if information does have to be updated, in order to keep track of recently used pages and we want to keep it as simple as possible. So, this very simple scheme that I am going to describe, maintains only one bit of status information with each physical page. So, among all the pages that are candidates for replacement, for each of those pages in its page table entry, keeps track of one bit of status information. Remember that LRU was going to maintain a time stamp, which could be a large number of bits of information, and then it had to compare all the time stamps in main memory.

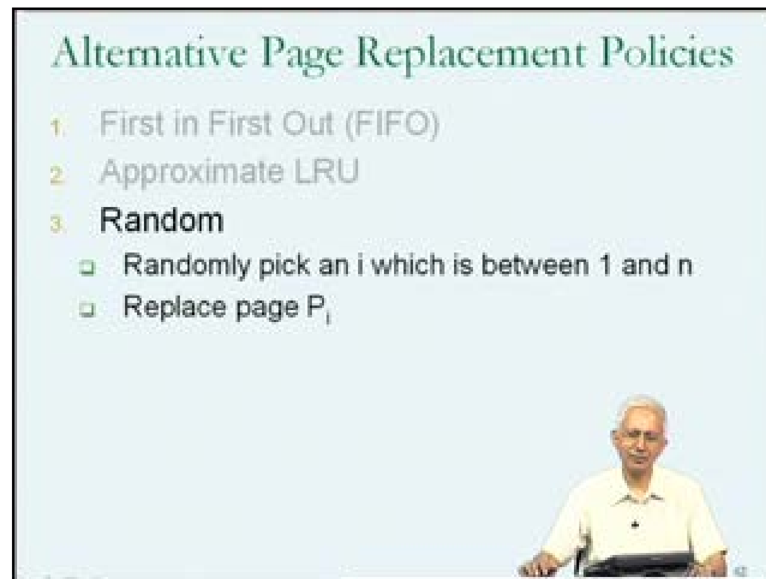
So, the comparison cost is going to be eliminated by having only one bit of status information, associated with each physical page. Now, how do we update this bit of status information? Obviously, it will have to be updated every time, a page is referenced. So, the idea is, initially all the pages, when they, when a page comes in, all the pages will have their bits set to 0. When a page is referenced, its bit can be set to one. So, a bit being set one is an indication that the page has been referenced sometime in the recent past. Now, after this has gone on for some time, it may happen that all the pages have their bit set, and therefore, there may be no information about pages which have not been referenced in the recent past.

Therefore, an additional feature of this policy, might be that after some time, every once in a while, we might reset all of the bits to zero. So, every once in a while, and that could be a parameter of the replacement policy, just reset all the bits to 0. How do we identify a page for replacement? Basically, among all the pages, which have a bit, this particular bit equal to 0, any one of them could be selected as a candidate for replacement.

Because among all the pages, which have not been reference in the recent past, all of those pages will have their bit set to 0, and only the pages which have been referenced in the recent past, will have their bit set to one. Along with this periodic resetting of all the bits to 0, in the sense, are being a reasonable approximation to LRU. One can improve the nature of this approximation by keeping a little bit more status information. For example, two bits of status information used in slightly different ways.

So, this raises the idea that one could approximate LRU possibly at with extremely low resultant over head, but reasonably, approximately to the principle of locality of reference, which might make it a feasible option, as far as incorporation into a page fault handler of an operating system is concerned. I am going to mention one more possible alternative page replacement policy, which is also sometimes used and that is what is known as a Random place replacement policy.

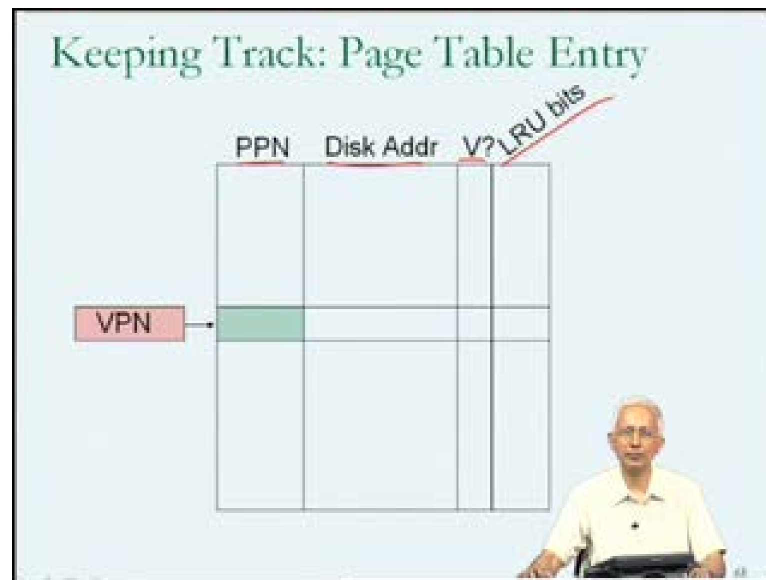
(Refer Slide Time: 09:00)



Now, the idea of the random page replacement policy is that when the time comes to pick from among the page one through page n which are in other words all the pages which are currently in main memory. The operating system could arbitrarily pick or randomly pick any one of them, and this turns out and studies have shown is not that bad a policy. It does not directly seem to have any relationship to the principle of locality of reference, but it does in practice work out not too badly. It has even lower over heads than the approximate LRU and if therefore, if the over head of the decision making is important consideration in this random might in fact, be a superior policy to be taken into account, preferable to approximate LRU.

So, you may in fact, find certain scenarios where random are the preferred policy. In other words, let the operating system is arbitrarily decide, you may think of this as by tossing a coin. So, that is what I mean by a random decision a multi-phased coin.

(Refer Slide Time: 10:14)



Now, how does the operating systems page replacement policy affect the information that is stored in the page table entry? From what we have seen, in order to keep track of exact LRU information, a time stamp or a stack had to be implemented and kept track of. Therefore, if I was talking about a situation where there was an operating system that had LRU as a space replacement policy, then the time stamp would have to for each of the pages, which is currently in main memory a time stamp would have to be remember and updated and this could be kept track of possibly in the page table entry, in which case the LRU bits or the time stamp would be present there.

On the other hand, if the operating system is implementing a two bit approximation to LRU, then two bits of information relating to page replacement might have to keep track of and they could conceivably be kept track of along with every page table entry.

So, we could at this point assume that in addition to the physical page number, the disk address, and the valid bit, the page table entry may contain some what I will call LRU bits, in other words some information, relating to the page replacement policy.

(Refer Slide Time: 11:25)

Page Fault Handler
1. Identify slot in main memory to be used
2. Get page contents from disk
3. Update page table entry
- Problem: The victim page identified by the page replacement policy might have been modified while it was in main memory
- It cannot just be overwritten by the incoming page
  - but must first be copied back to disk
- Optimization: Keep track of whether it has been modified while in memory

Now, you will going back to what the page fault handler has to do, the page fault handler has to identify a slot in main memory to be used and we have seen a little bit about that. We see that sometimes in order to identify a slot in main memory, there may be the need to create an empty slot in main memory, which is where the page replacement idea came into the picture, and subsequently the page may have be fetched from disk and the page table entry updated which were relatively easy tasks, but as it happens we are not yet finish with the first task of the page fault handler.

Now, let us just think about this little bit. By the end of the first task, the operating system page fault handler is at a point where can just fetch the page, which is required or copy the page, which is required from the disk into the slot, which has been freed due to the page replacement activity.

But it could happen that the page, which has been identified by the page replacement policy, might have been in memory for a long time and might have been modified while it was in memory. In other words, every time the processor execute a store instruction or every time a program modified a variable, which was on that page, the modification would have happen to the page in memory and if the page has been modified, while it was in memory, then it cannot just be over written by the incoming page.

We cannot just copy the new page, which is going to come in response to the page fault having occurred and overwrite the page, which was there before. It will actually be necessary to copy the page, which was identified for a replacement, because it is different from the copy on the disk. So, it has to be copied from the main memory back to the disk. When the page was modified, whenever it happened, the modifications were done only in main memory and therefore, the page became the copy of the page in main memory became different from the copy of the page on the disk. Hence, the copy of the page on the disk is no longer correct and if we lose track of the copy of those changes, then the program will not execute correctly. The variable x, which was modified only in main memory, will not have its correct value on the disk, and the next time this page is fetched from the disk to the main memory, it will have the old value of the variable x. Therefore, it becomes necessary to first copy the page from the disk from the main memory back to the disk.

Now, the thing to note here is that this means whenever there is a page fault; there will be times when the page has to be copied back to the disk. Every time, there is a page fault they will be a need to copy a page from the disk to the main memory. That is the page, which is required by the processor, but it will be necessary to copy a page from the main memory back to the disk. In other words, the replaced page, only if the page had to be replaced. Remember that it was possible that they may have free pages in the main memory, in which case, page replacement did not have to happen at all. There was an empty space in the disk on the in memory, which could be used to copy the page that is required.

Now, in this particular case where the free list was empty and the page have to be replaced, once again it is not always necessary to copy the page from main memory into the disk, as part of replacement. It will be necessary to do that only if the page has been modified. And you should note that since the amount of time to do a disk access is orders of magnitude more than the amount of time to do a memory access, it will be in the interest of the system of the execution time of your programs, and the activity of the system to try to keep the number of times that disk access is happen, as low as possible.

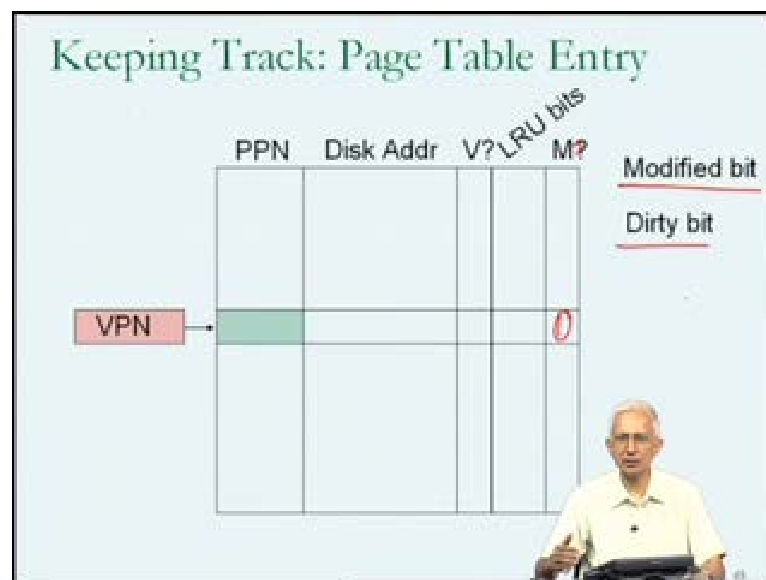
Therefore, if there is a way, by which the writing of the dirty page, the page which has been modified in main memory back to the disk, down to the minimum cases necessary, then that would be good for the execution time of programs and the efficiency of the

system. Now, I will refer to this as an optimization, because the operating system could actually keep track for each page of whether or not it has been modified, while it was in memory. And if the page which has been replaced, has been modified while it was in memory, then and only then needed be copied back to the disk as part of step one of page fault handling.

If the page has not been modified while it was in main memory, it can safely be over written, by the incoming page, because the copy on the disk is the correct copy. There is nothing new that has to be done as far as the updating of that page is concerned.

Now, the question now becomes how can the operating system keep track of whether or not a page has modified while it was in main memory, and the answer is whenever a store instruction is executed, which is the way that variable gets modified, the consequence of doing the store should be to note down the fact that the page containing that address has been modified while in main memory.

(Refer Slide Time: 16:24)



And this can done with a single bit, in other words, as part of the page table entry of every page, a single bit could be used that is why I refer to it as the M question mark, meaning a single bit M typically refers to the word modified.

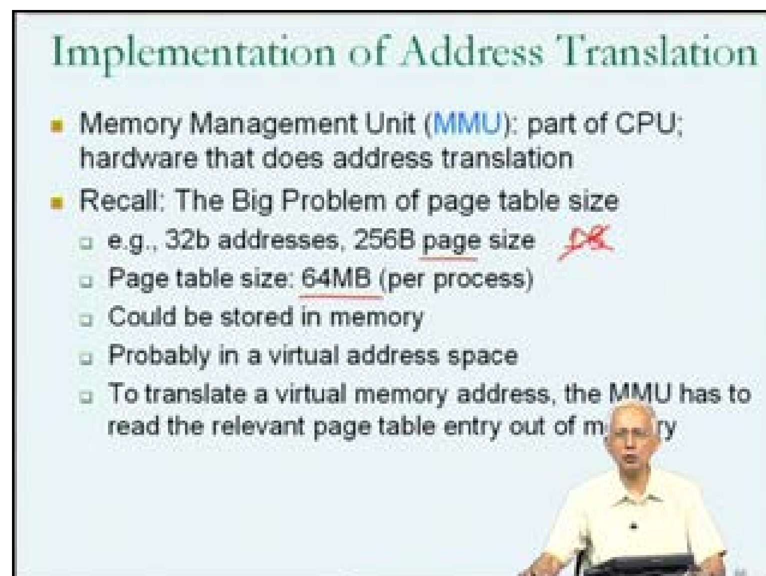
So, the bit will be set for a page, if while that page was in memory, a store instruction had executed to a piece of data which is on that page. So, if there is a page, which has

this modified bit equal to 0, then when that page is replaced, it need not be copied back to disk. That is the optimization that is achieved by having the modified bit. The modified bit is sometimes referred to as the dirty bit. I, inadvertently referred to it as a dirty bit, few seconds back.

So, we understand that if a page has if a piece of data on the page has been modified while it was in memory and so we understand why is called the modified bit. It is called the dirty bit because the fact that the page has been modified in memory makes it different from the copy on the disk, and that is an deviation from the correctness of data contained on the two pages and is sometimes refer to its dirtiness.

But you find either M or D to indicate that this is a modified bit, a bit which indicates whether the data on that page is different from the data on the disk. So, we are seen that the amount of information that has to be kept track of, by the operating system, in connection with each of the virtual pages, is increasing and this is happening because of the complexities of managing the show behind the scenes.

(Refer Slide Time: 17:54)



Now, coming back to address translation. We understood that all of this activity was happening, because the page table was being maintained largely to as a facilitate address translation. In other words, facilitating many programs being in memory at the same time, each of the programs assuming that it can use its virtual addresses, addresses in the

range 0 to n, 2 to the power n minus 1, while in reality the main memory addresses are potentially different and hence a need for translation, to protect one program, one process from another.

Now, we had referred to the fact that the address translation itself is done by a piece of hardware call the memory management unit or MMU, and that this is part of the CPU. It is a piece of hardware that does the address translation; the address translation itself is quite easy to do.

Now, let me just refer you back. The address translation itself is quite easy to do, in the sense that all that the piece of hardware has to do is recall that the virtual address. We now view as a virtual page number, the most significant bits of the address are a virtual page number, the least significant bits of the address are the offset within the page, and in order to do that address translation, all that the memory management unit has to do is get the corresponding page table entry, and for that particular virtual page number find out what the corresponding physical page number is.

So, that means, look up the virtual page number in the page table, and the second thing that the MMU has to do is, construct the physical address, by taking the physical page number and appending to it the offset bits, the least significant bits from the virtual address.

So, the task of address translation which the hardware does is actually very simple. It is a look up into the page table, followed by a concatenation of bits, from the virtual address along with bits from the page table, the physical page number.

Now, let me just refer you back to the slide which much earlier where we have talked about the big problem of page table size. You will recall that when we looked at an example of 32 bit addresses, in other words, virtual addresses are of size 32 bits, physical addresses are of size 32 bits, and we were assuming that the size of each page is 256 bytes. We saw that the address, the size of the page table and remember this was one of the favorable examples; the size of the page table was 64 mega bytes per process. We had said that this is acceptable, because the size of main memory today is gigabytes. Therefore, we could comfortably accommodate many page tables in the main memory.

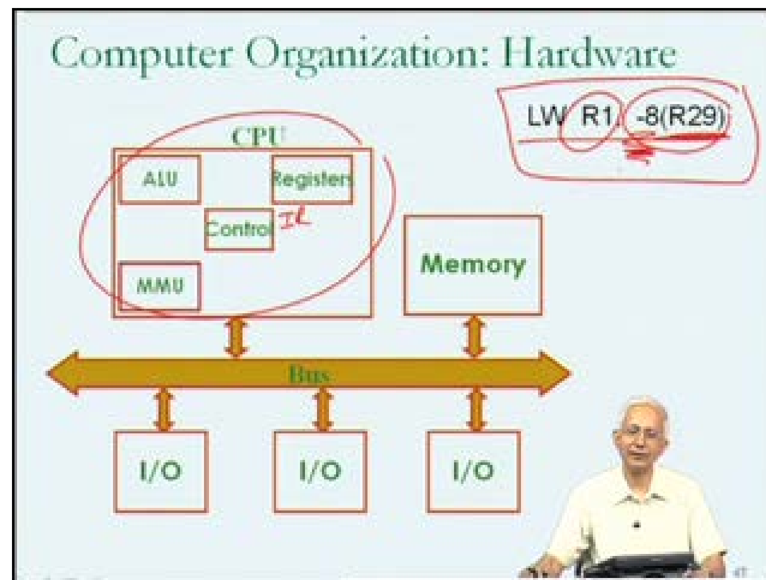
We had come down to 256 bytes from one, we had come up to 256 bytes from keeping track of one piece of translation information for each byte, where we had seen that was clearly infeasible, and that was the motivation for talking about pages at all, in other words, making the granularity of address translation much larger. But let us just think a little bit about where we are at now. Even now, the size of the page table is 64 mega bytes per process. So, if there are 100 processes currently in memory then we have 100 such page tables which must be available.

Now, one possibility is this page tables could be stored in memory, and we seen that there will be enough space in memory for this purpose. But, it may not be possible to store all of them in main memory at the same time, because they occupy so much space. If there are 100 processes then us talking of 6400 megabytes of memory, which could be a reasonably substantial fraction of the main memory size.

So, more likely, it would be the case that the page tables would be stored in a separate virtual address space, may be in a virtual address space that the operating system uses. So, that at any given point in time, just like for the individual processes, if the operating system has its own virtual address space, then some of the contains of the page tables could be present in the main memory. That could be decided based on the similar considerations along the lines of what were done for ordinary user process virtual address space.

But even then, in order, to translate a virtual memory address into a physical memory address, the MMU, the memory management unit, the part of the hardware that does the address translation, would have to read the relevant page table entry, out of main memory. Remember, the address translation is going to happen every time an address is referred to by the processor.

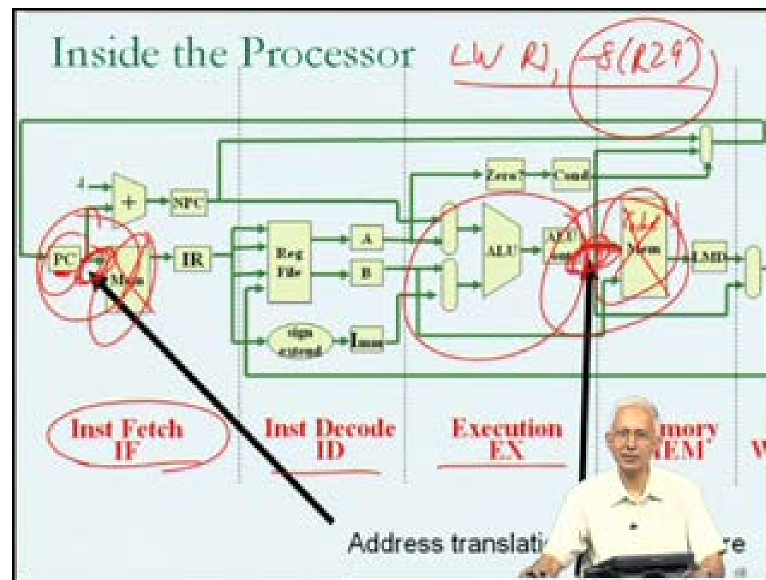
(Refer Slide Time: 22:20)



Let us just think about this a little bit. So, we have this picture of the computer, the instructions are being executed over here. Everyones in a while it might happen that a particular instruction, for example, the current instruction, which is being executed, might be instruction which does a memory access such as load word R 1, minus 8 R 29.

So, the effective address is going to be calculated by the hardware, and I move to the next slide (Refer Slide Time: 22:49). In our looking at the inside the processor, we know that for an instruction like this load word R 1, minus 8 R 29, the virtual address has to be generated. the virtual address is of The memory address from which the variables value is suppose to be loaded into R 1 that is the virtual address, the virtual address has to be generated by adding the contents of R 29, which is the base register to the displacement which is minus eight.

(Refer Slide Time: 23:11)



And when we looked in detail at inside of the processor, we realize at that addition was going to happen in this part of the processor. So, after the load word instruction had been fetched and decoded, when it comes into this execute hardware that is where the effective address gets calculated, the actual virtual address, and that virtual address is what is going to memory over here, which means that the address translation would actually have to happen between the calculation of the virtual address, and the use of the virtual address, this could be the cash data cash memory, but between the generation of the virtual address and the calculation of the virtual address.

So, we were actually talking about the memory management unit during the translation somewhere in this part of the processor, and you will bear in mind that this is not the only kind of memory access that might happen as far as instruction is concerned. This self same instruction, load word R 1, minus 8 R 29, had to be fetched and in order to fetch the address, in order to fetch the instruction, it is address would have to be so the address comes out to the program counter.

So, this is the part of the hardware where the instruction was being fetched. In order to fetch the address, its instruction had to be sent to memory, but the instruction address, which is inside the program counter is once again a virtual address, and therefore, once again a translation has to be done. So, here once again the MMU has to translate the

virtual address into a physical address and then that physical address could go to the cache or the instruction cache, whatever it is.

So, in the execution time of this one instruction, it is conceivable that like this load word instruction, this address translation would have to happen two times. Once for the fetch of the instruction over here, and once for the fetch of the data over here, therefore for this load word instruction, we talked about a cache memory, and we argued earlier, when we are talking about the implementation, we said we do not have to worry about accessing memory over here or accessing memory over here, because they are these things called Cache Memory. And because of cache memorys we do not have to see the 100 nanosecond latency of the memory, we can access the data most of the time in one nanosecond, and we discounted the fact that load word instruction had to access data or the fact that the load word instruction had to be fetch from memory itself.

But now we have come to a situation where we realize that in order to do the address translation over here (Refer Slide Time: 25:33), and over here, the MMU has to refer to the page table entry, which is in memory. It is once again, every instruction is going to require a memory access, just to access the address translation information. This is clearly not going to work. We had to incorporate a notion, the assumption about cash memories, just to make the hardware a possibility. Otherwise, everything would be swamped by the 100 nanosecond delay to get something out of memory and suddenly we are back in that position again.

(Refer Slide Time: 26:04)

Implementation of Address Translation

- Memory Management Unit (MMU): part of CPU; hardware that does address translation
- Recall: The Big Problem of page table size
  - e.g., 32b addresses, 256B page size
  - Page table size: 64MB (per process)
  - Could be stored in memory
  - Probably in a virtual address space
  - To translate a virtual memory address, the MMU has to read the relevant page table entry out of memory
  - Hardware must provide page table entries [illegible] than that (most of the time)

inside the MMU

Therefore, something has to be done to correct this problem with address translation. As for the problem is that to translate a virtual memory address, the MMU has to read the relevant page table entry, and the page table entry is present in a page table, and the page table is potentially 64 mega bytes in size, and there are lot of page tables. So, we may be able to have some in the system, but at best they are going to be in main memory, and therefore, the MMU is going to have to make a main memory reference, every time it has to do an address translation. This becomes a problem.

We have to understand solutions to this problem. Right now, the solution to this problem is once again to make an assumption, that this is actually possible, to have inside the MMU, a piece of hardware, something like the cache memory that we talked about earlier, a special piece of hardware inside the MMU, which will be able to provide the page table entry that is required at least most of the time.

So, this sounds like a cache like idea, we do not know exactly how the cache memory works, but here once again, we are talking about the capability for a piece of hardware to provide some piece of data, at very high speed, most of the time, rather than having to get it out of memory and therefore, we suspect that the design of this special piece of hardware inside the MMU, must be similar to the design of the cache memory, and that when we study about cash memories, we will be understanding something about the design of this piece of hardware.

(Refer Slide Time: 27:27)

Implementation of Address Translation

- Translation Lookaside Buffer (TLB): memory in MMU that contains some page table entries that are likely to be needed soon
- Recall: Cache memory contains data/instructions that the CPU is likely to need soon
- If the required page table entry is NOT present in the TLB, then the MMU can do the translation fast
- Otherwise: TLB miss
- Must be handled, possibly like the OS handles a page fault

Now, this piece of hardware inside the MMU is known as the Translation Lookaside Buffer frequently abbreviated as TLB. So, it is some form of hardware, including memory, inside the MMU that contains several page table entries. In particular, page table entries that are likely to be needed sometime in the near future.

So, that most of the time, the MMU will be able to do it is address translations, just using the page table entry copies, which are available in the translation lookaside buffer. Now, if you look at this name carefully, we understand why the word translations is there. It contains information relating to address translation. It is called lookaside, because it is the shortcut to having to look into main memory for the same information.

The word buffer stands for the small amount of memory, which is used for remembering things somewhat temporarily. Now, we have seen that caches do something similar. So, as I had mention earlier, we suspect that the design of a TLB, I will use that abbreviation, it is likely to be similar to choose a design of a cache. Now, what about this situation which I will talk next, it is possible that the page table entry which is required by the load word instruction is present in the TLB, and in that case the MMU, the memory management unit hardware, can do the translation extremely fast. The translation itself as I seen is very simple to do if the page table entry is available.

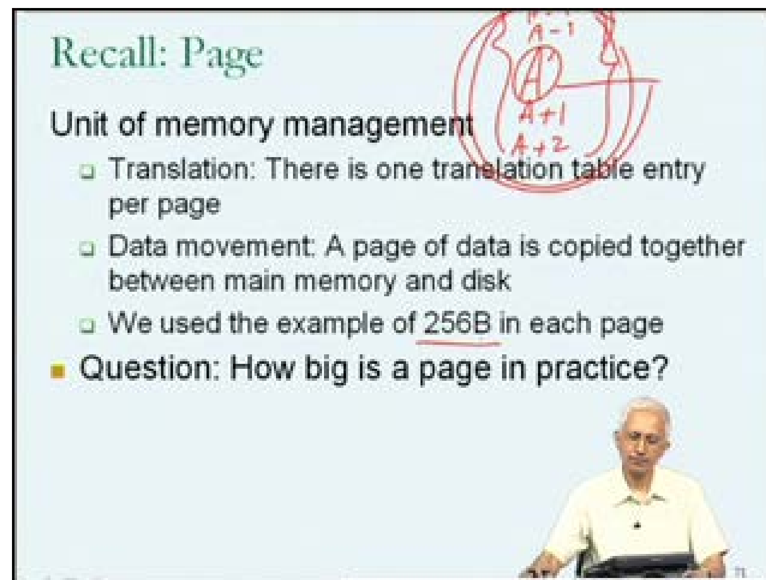
What is the page table entry is not available? In other words, what if the page table entry that is required for the current address translation is not present in the TLB? This is again the page fault that we had seen in the case of the virtual memory, where the page that is required by the processor is not present in main memory and therefore, had to be read out to the disk, and this situation that is described here is technically refer to as TLB miss. It is the situation where the processor has generated a memory reference, load word R 1, minus 8 R 29, address translation is being attempted for that minus 8 R 29, which might be the in address, let us say hex 1 0 0 0.

So, 1 0 0 0 has gone to the MMU, the MMU has looked up for the corresponding page table entry, inside it is translation lookaside buffer, and it finds out that the translation information is not available there, for whatever reason. This is called the TLB miss, and once again, this cannot just be ignored. It has to be handled, therefore, they has to be a handling of the TLB miss, as a result of which, at the end of the handling of the TLB miss, the required page table entry must have been copied from the page table where it was in memory into the TLB, so that the MMU can then go ahead with a translation.

So, that is the and this would run along line similar to what we saw for the page fault handling, but it certainly must be handled, it cannot just be ignored. Now, the handling of the TLB miss could possibly be done by an operating system handler, but there are some situations where the handling of the TLB is built into the hardware. In other words, the part of the MMU has capabilities of handling a TLB miss.

It is in some situations, some pieces of hardware that is what happen. So, we understand how address translation happens, we understand that there is no need to access the page table entry out of main memory, as part of the address translation, rather in most cases the page table entry is available inside this TLB, which is a small amount of memory present, inside the memory management unit, which itself is a piece of hardware present inside the processor.

(Refer Slide Time: 31:09)



Now, there are two other issues, which I would like to talk about in connection with paged virtual memory, and one relates to the notion of the page. You will remember that the page, if you look back, the page is the unit of memory management. The operating system does not manage memory on units of bytes or words; it does not maintain address translation at that level. Rather, it maintains and manages virtual address space and physical address space in units of pages, and we see this in the fact there is a piece of translation information for each page, and also that whenever we talked about moving data between the disk, which is where all the virtual address page is stored and main memory where it is temporarily held, The movement of data is always in units of pages.

There is no question of moving half a page or quarter of a page or only a byte from disk to main memory, rather an entire page is copy from main memory to disk often disk to main memory. So, the page is both a unit of translation and the unit of data movement between main memory and disk. Now, the question of how big a page is of some relevance to us and the reason that is some relevance to us is because we have heard about the principle of locality of reference, which talks about the neighborhood of an address A.

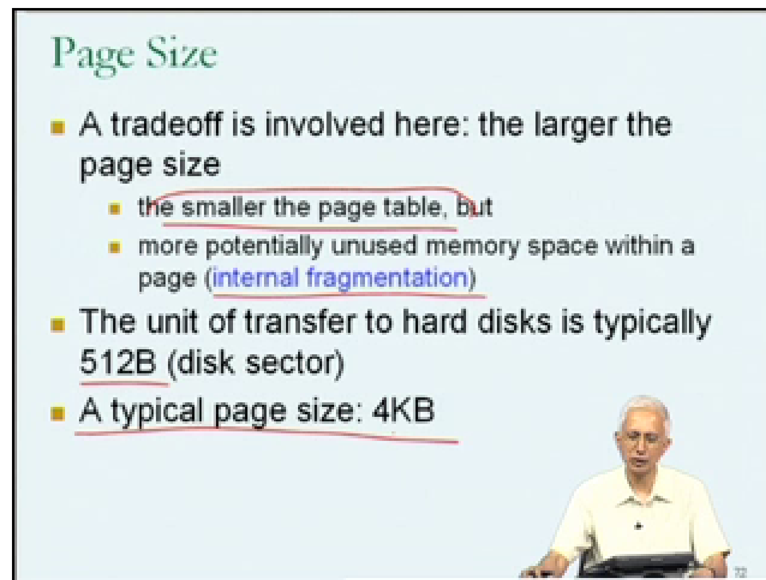
We talked about how according to principle of locality of reference if address A is reference a time t, then it is likely that address A minus one, and address A plus one, and address A plus two, and address A minus two, in other words, the neighborhood or the

neighbors of the address A are referenced in the near future, as far as the time, at which the reference to a took place. Therefore, if the neighborhood, as far as your program is concerned, is much larger than the size of a page then this is something that you would have to be aware of.

So, since the concept of principle of locality of reference has this notion of neighborhoods, it would help us to know little bit about what the size of pages. For example, if I now told you that the size of pages is 64 bytes, then you might be a little concerned, because it might happen that the programs that you write typically have neighborhood, which are larger in 64 bytes and therefore, the operating systems perspective on locality of reference is not in sync with your programs requirement as far as locality of reference is concerned. In our example, I had used 256 bytes as the page size.

For the purpose of the example, we need to understand a little bit about what considerations are used in deciding actual page size is. So, the question of how big is a page in practice and you could actually get an answer to this for your own system quite easily on the system itself. More system has a way for you to find out the page size using mechanism that might be called as get page size, for example. The very fact that the system has gives you the capability of finding of the page size, means that it is something that programmers might find useful in the writing of programs. Writing a program, so this program can check what the page size is on the machine where it is running, in case it is of relevance to the programmer.

(Refer Slide Time: 34:24)



Now, let us talk about this little bit. What are the considerations under by which the person who makes the page size decision arrives at a number? Now, we look at from two from two prospective. What is the advantage of having small pages? What is the advantage having big pages? Is there is any, in each of these cases? Now, what would be the advantage of having a large page? If you think about it a little bit, if we have very large pages, then we will have less translation information. For example, if I made in rather than talking about 256 byte pages, if I had one mega byte pages, then very clearly they would be one piece of translation information on each mega byte of memory, and therefore, there will one page table entry only for each mega byte of memory and therefore, the size of the page tables would be much smaller, substantially smaller.

And we have seen that the page table size is a concern because the larger the page table the more the likelihood that the most storage will be required to store it, and also the more the likelihood of the TLB misses. TLB misses are the situations where the MMU cannot translate a page. Therefore, large page tables can result in performance problem as far as your program is concerned, more and more TLB misses, which have to be handled before the program can actually do the memory reference.

Now, for a large page, the page table will be smaller, but the negative side of things is that this potentially more unused page within the page, as far as your program is concerned. Think about it in this way. Let suppose that you had a situation where there

were one mega byte pages and you had your program and you know that the text, in other words, the instructions of your program are far less than one mega byte in size. The instructions of your program might require only 30 or 40 kilobytes of storage. Now, if the page size is one mega byte, then smallest unit of granularity, as far as the text of your program is concerned is going to be one mega byte.

And this going to be one page, which would be use for the instructions of your program and within that page if your instructions of your program occupy only 13 kilobytes, then the remaining almost one mega byte of storage is going to be unused, is going to be lot of wasted memory, wasted main memory. Remember, this is the memory, which is going to be waste inside the main memory. In addition, to be wasted on the disc, this wastage of memory having unusable or unused memory space within a page is what is known as internal fragmentation. So, the larger your page size is, the more potentially the internal fragmentation, the inefficient use of main memory. However, larger the page size, the better you off from the prospective of page table, so there was a tradeoff here and somewhere in between is going to be the typical page size.

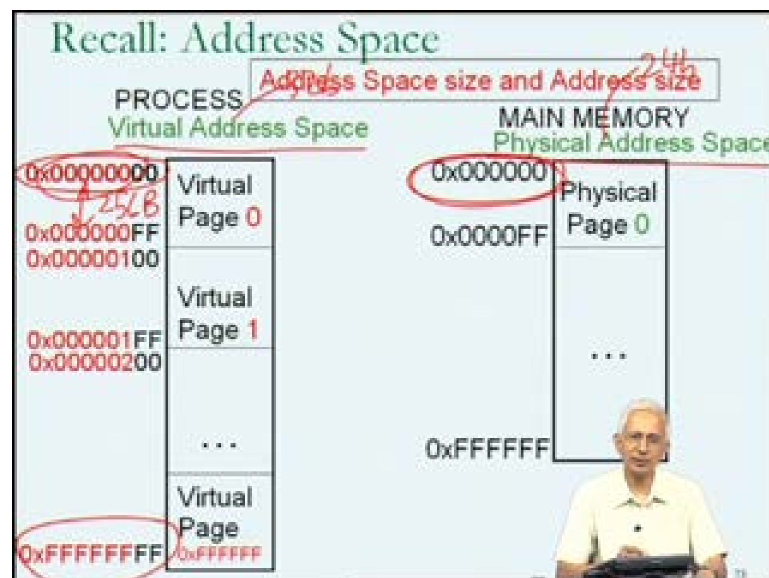
So, it is unlikely to be mega byte, is unlikely to be bytes. The other consideration which would come into play here is we know that in the implementation of paging, data is going to be transferred between main memory and the disc. Whenever a page has to be read from this disk, it will be read from disc to maim memory. Whenever a page has to be return back to the disk, in the event of a modified page being replaced, it has to be copied from a main memory to disk. Therefore, the question arises is what are disk like. What is the unit of transfer to hard disks? Now, it turns out to the unit of transfer of data to hard disk is not bytes. It is not mega bytes, but it is in the units of approximately kilobytes.

The typical sector size of the unit of transfer to a disk is something like 512 byte, in some cases, possibly one kilobyte. Now, why is this consideration here? We know that if the unit of transfer between disk and main memory is 512 bytes and we use a page size of one mega byte, then the very large number of disk access is will be required, to fetch a page from disk to main memory. So, this is suggesting that very large page sizes would be problem for the disks of today. On the other hand, if I use a page size, which is let us say a few kilobytes, then the number of disk accesses required to fetch a page, a copy of the page from disk to main memory will be very few.

Hence, today you may find out that there is a typical page size is few kilobytes. Frequently occurring number would be something like 4 kilobytes, sometimes 8 are 16 kilobytes, small number of kilobytes. This is also something, which may be reassured of from the prospective of the neighborhoods of your program are concerned.

If use this kind of number as a guide line, we realize that a small number of disk access is would be all that are required to fetch page from disk into main memory, and that the number page table entries may be more than it would have been with gigantic pages. As a compromise, sometimes happens operating systems today, maintain pages at two levels. They may actually have a level of pages called Super Pages, which may be of mega byte in size or even larger than mega byte in size, to manage multiple smaller pages with in them, just once again to play the tradeoff between the page table size and the unit of transfer between the disk and main memory. But, from our perspective from this point on, we could work under the assumption of page sizes of approximately 4 kilobytes. As I said, in real system with you work with, actually you find out what the page size is and if it is substantially different that is the kind of number, you should bear in mind when thinking of the size of a page.

(Refer Slide Time: 39:49)



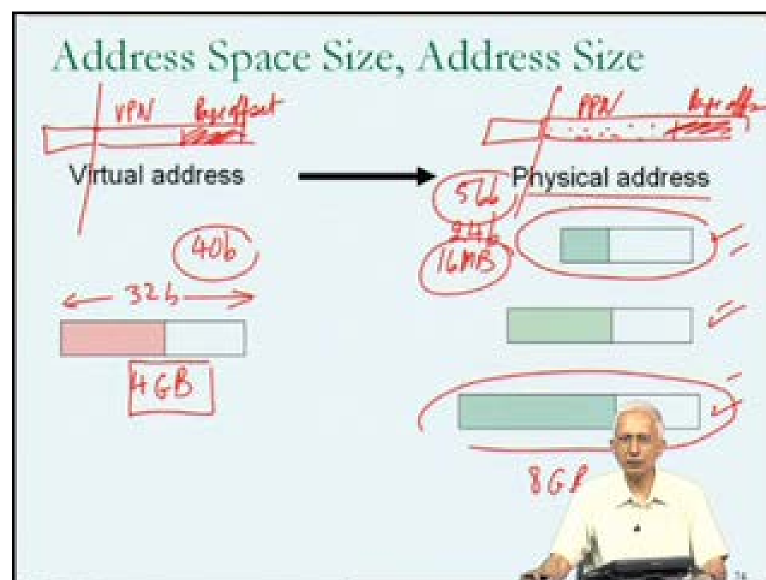
Now, just relating back to what we had seen when we talked about address translation. You will remember that we were talking. In this diagram, I am actually still showing you a diagram in which the assumption is being made that the pages of size 256 bytes. But

remember that more realistically now we should be thinking about pages of size about 4 kilo bytes, for reason that we saw in the previous slide.

Now, the reason that I bring up this size is that you will note that we were to use the terms virtual address space and physical address space, and in this particular diagram, I had made the odd assumption that the virtual addresses were of size 32 bits and that the physical addresses were of size 24 bits. So, the virtual address space of the virtual address was of size 32 bits and the physical address was of size 24 bits, and at that time I had just said to let us make this assumption without actually motivating what this might be is the case. So, this actually raises an interesting issue, which we should delved into a little bit further, that is the size of the address spaces.

In this diagram, I had make this assumption that the virtual address space is bigger than the physical address space, and this may have raise in your mind the question of cant they be the same size or even cant the physical address space be larger than the virtual address space? So, let us just address that issue a little bit, because in practice the system that you deal with, you must be aware that the consideration like this may affect the way that your programs are execute and so we will talk a little bit of our address space size and address size. Here, remember the address size was 32 bits, virtual address size was 32 bits, and the physical address size was 24 bits.

(Refer Slide Time: 41:36)



Now, remember that the size of the address space is going to be determined by the size of the address. Here, the size of the address space went from 0 up to the maximum possible address that was possible, and the maximum possible address is going to be determined by the size of the address.

Therefore, in talking about the size of address space, it is sufficient if we talk about the sizes of the addresses, and for any particular virtual address size, let me assume that the virtual size is 32 bits, whatever it is. They are three possibilities. The physical address could be smaller; it could be same size or could be a larger. And we need to understand how each of these three situations might arise and try to figure out whether or not they are feasible.

Now, the first question that might arises, why would it be necessary to sometimes have situation where the physical address space? In other words, the size of the physical address is less than the size of the virtual address, such as in the first diagram or in our example the physical address was of size 24 bits and if you thing about this little bit you will realize that if you have a 32 bit virtual address space; that means, that the size of the address space is 2 to the power of 32 minus one, which is that for, if these are byte addresses, then we are talking about an address space, which can address 4 gigabytes of memory of virtual addresses of virtual address space. So, the address space sizes are 4 gigabytes.

These two 24 bit, that we have over here, corresponds to something much smaller. In fact, its talking about something which is of size 16 megabytes, and in the not too many years ago, if you bought a laptop you may have been told that for the amount of money that you have, this laptop can have 16 megabytes of memory. So, we realize that the amount of memory that a system has could well relate to the cost of the system.

Today, you can buy a laptop with 2 gigabytes of memory, but again that may be more expensive than a similar laptop, with one gigabyte of memory, and clearly a laptop with two gigabytes of memory cannot address data with address physical addresses more than the 2 gigabytes. So, that is the capacity of that main memory and therefore, the size of the physical address must clearly be such that it is commensurate with the amount of memory that present.

Therefore, to some extent, you might observe that the size of the physical address will relate to the size of the physical memory, the size of the main memory that is present. The cheaper the system, it is possible, the less the memory there is, since memory contributes substantially to the price of the system. But at the other end you may ask, there is this possibility of the address space size being larger, the physical address space size being larger than the virtual address space, in other words, I can address four gigabytes of virtual addresses, but I may be able to address 8 gigabytes of physical addresses.

And with this makes any sense and once again the answer might be, if there is enough, if one is able to put 8 gigabytes of memory into the system, and the hardware allows one to have 8 gigabyte addresses, then you can only be the benefit to the programmers. It can only be the benefit to the people who use the system, since the number of virtual pages, which can be present in the memory at any given point in time, can be more. So, none of these scenarios would actually be viewed, has been ruled out, but one should bear in mind that the actual performance of the system will depend on the size of the physical address space, in terms of the speed with which the average memory access will take place, the frequency with which page falls will take place etcetera.

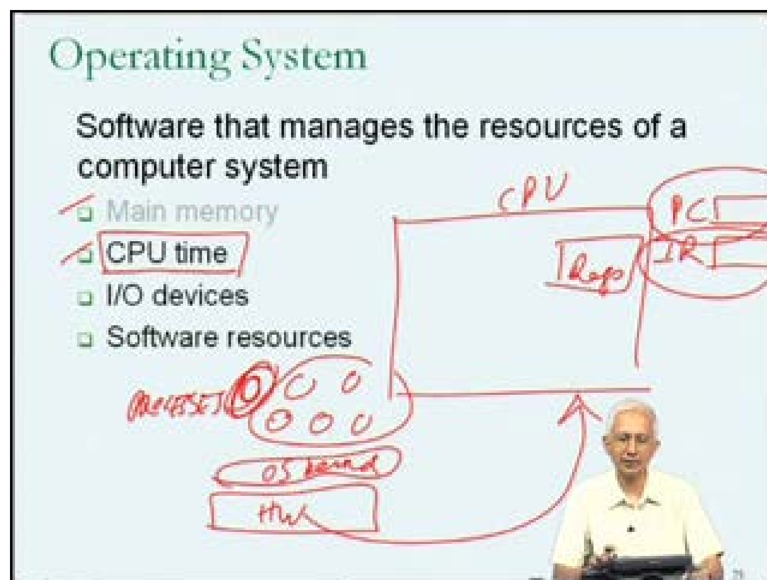
Now, one aspect relating to virtual address and physical address that I did not talk about, I would like to address, right now, and that is we have talked about how the virtual address is made up of virtual page number and a page offset. The physical address is made up of the physical page number and the same page offsets. So, these bits are identical to these bits, and address translation just replaces the virtual page number bits by the physical page number bits.

But the question of whether any other information might be present inside the addresses, as far as internal to the processor is concerned. I raise this issue, because some of you may actually look at hardware manual, and you may find out that the virtual addresses on the system, are described as being 40 bit virtual addresses. The physical addresses on the system are described is being 56 bit physical addresses. So, it would seem that the picture that we are getting as programmers or from the prospective of the operating system, is incomplete and the some additional information might be added for hardware purposes, to argument the use of the information, by the translation and the management

mechanism, but since this is not of relevant to us as far as the impact on our program is concerned.

But it might have something to do with management at the level of multiple programs, it is not something the need concerned us, and therefore, we need not try to worry too much about information that you read in the hardware manuals about the actual size of these addresses. The actual size of these addresses could be more. So, on a 32 bit machine you may learn that the addresses are kept in a 40 bit register for example, or that the size of the physical page number inside the page table entries, such that the address would be 56 bits in size. So, these are not considerations that need weight too heavily with us.

(Refer Slide Time: 47:09)



So, with this we have a reasonable understanding about how address translation works and this is a reasonable time to try to move on to our next topic within operating systems. You recall that the operating system is software that manages the resources of a computer system and there are many aspects of this that we have to look at. The first aspect which we have now completed is a picture of what happens both behind the scenes and specifically within the operating system, in terms of the key decisions that it has to make, in terms of sharing physical or main memory, among the many programs that are in execution on a computer system, at a given point in time.

Now, we will next take a look at CPU time, and as mentioned the reason that this the topic comes into our discussion is, we had seen that within the CPU, there is only one collection of registers, and in particular there is only one program counter register, there is only one instruction register. The program counter is an important register; it contains the address of the instruction, which is currently being executed. The instruction register is a very important register; it contains the bits of the instruction that are currently being executed, which raise the question of what does it mean to talk about hardware and an operating system and many process?

This is a sketch of the diagram that we had seen earlier. This is the hardware, this is the OS kernel and these are the processes. So, they could be, as I said they could be 100 processes on the system. What sense does it mean to talk about 100 processes executing on that piece of hardware, at a given point in time, if the hardware is such that there is only one program counter? If there is only one program counter, and there is only one instruction register; that means, there only one instruction can be in execution at any given point in time, and very clearly that instruction can only be from one of those 100 processes and that evidently .the other processes are not actually running.

Now, all of this is what is happening behind the scenes as part of the operating systems CPU time management functionality, and we differ have to look at this in significantly more detail to get an understanding what is happening our program. We understand that our program, which is may be running as this process, may be sharing the CPU along with 99 other programs and that therefore, it might only get 100th of the attention of the CPU.

In other words, out of every hour of CPU time, it may get only one hundredth of that time, and that the operating system might it be making decisions, along the lines how to share the CPU time among the different programs, which are in execution, which might be of great importance to us, in deciding how to modify our program, in order to execute as fast as possible. Since, what is of concerned to us is not the well being of the other programs that execute on the computer system, but the well being of our program.

We try to modify of our program to run as fast as possible or occupying as little memory is possible. Our considerations in writing the program are always based on the well being of our program and that therefore, CPU time management is one of the critical aspects of

the operating system, which we need to understand in some detail, and we will start doing this in lecture 16.

Thank you.