

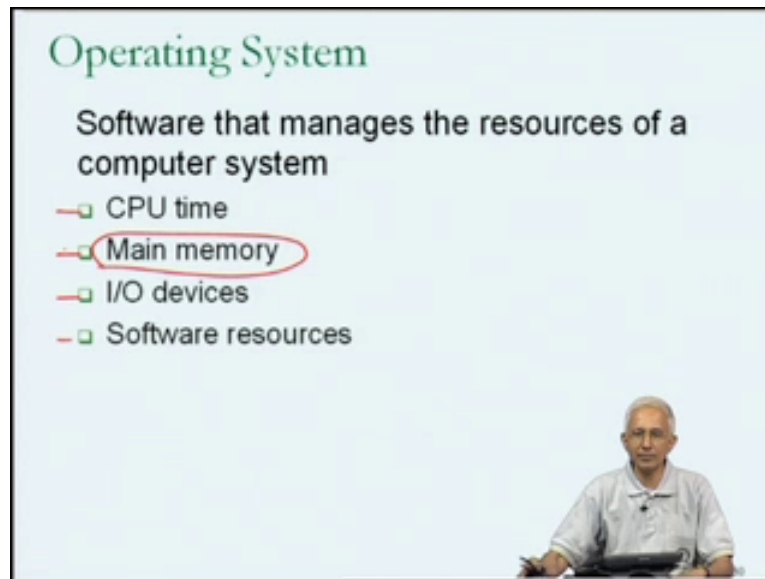
High Performance Computing
Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 03

Lecture No. # 13

Welcome to lecture 13, of our course on, “High Performance Computing”. As you will recall in the previous two lectures, we have started looking at the software side of the system, as far as, what is required for the execution of our programs. And, towards the end of the previous lecture, we had looked into the very important piece of software known as, the Operating System.

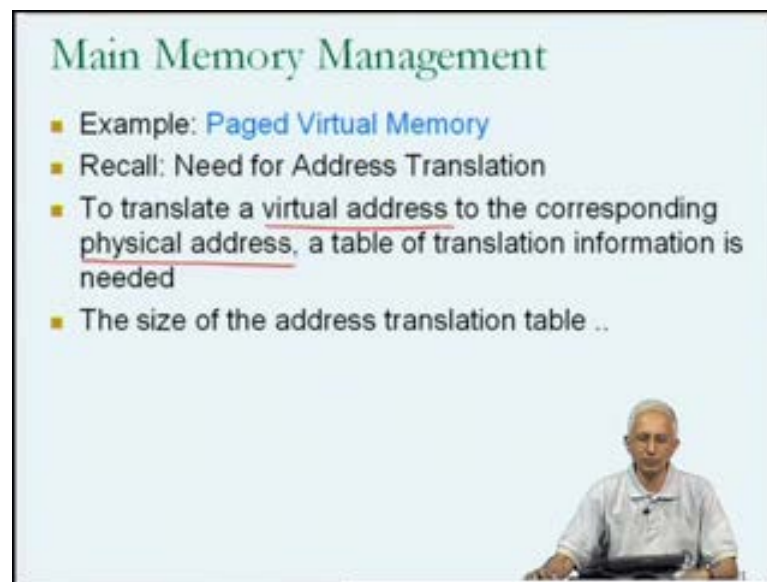
(Refer Slide Time: 00:39)



So, let just start off, with a brief recap from the previous slide, where we saw that the operating system is the critical software on a computer system. It manages the resources of a computer system, and so far, the perspective of what happens to your program, when it executes. Software System is clearly very important. The software system is managing the hardware and software resources. We had seen that we need to understand four aspects of what the operating system does: how it manages the CPU, the sharing of the

capabilities of the CPU among the different programs in execution, how it manages the main memory, ensuring that one program cannot access and modify the variables of another program for example, how it manages the I/O devices, such as terminals, monitors, disks, network, interface, cards etcetera. We will not be looking at all of these examples, but one. And finally, how it manages the software resources of a computer system, and the key concept here was that when a program is in execution, we have the concept of a process. The operating system manages the different processes, which are sharing the resources of a computer system.

(Refer Slide Time: 01:54)



Main Memory Management

- Example: **Paged Virtual Memory**
- Recall: Need for Address Translation
- To translate a virtual address to the corresponding physical address, a table of translation information is needed
- The size of the address translation table ..

The slide features a light blue background with a black border. At the bottom right, there is a small inset video of a man with grey hair, wearing a light-colored shirt, sitting at a desk and looking towards the camera.

So, our first look into operating systems is in the area of how the operating system manages main memory. And we are looking at specific kind of widely used operating system mechanism for managing main memory. This is known as Paged virtual memory. And I just remind you that it is necessary to have a mechanism to protect one program in execution from another. That is typically done through a procedure of address translation.

In other words, the program in execution will address its text, data, stack and heap, using may be addresses between 0 and some maximum address, based on the size of an address. But these addresses would not really be main memory addresses. They would be referred to as virtual addresses. Each process would have its own virtual addresses, and

these would ultimately be translated when the programs execute into actual main memory addresses, which are called physical addresses.

So, to do this translation, a table of the mappings between virtual addresses and physical addresses is necessary, and we understood in the previous lecture that unless we do this carefully, the size of the table, size of the address translation table, could itself become major problem.

(Refer Slide Time: 03:06)

Address Translation Table Size

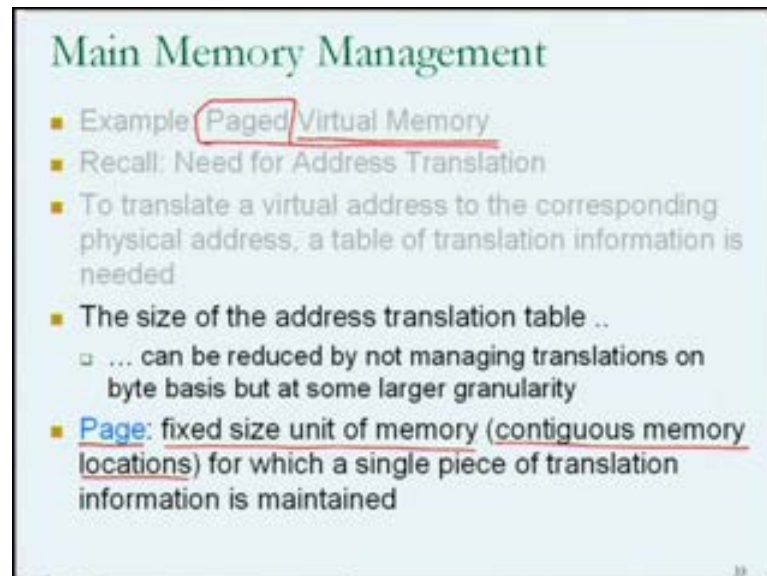
- Mapping from virtual address to physical address
- If there is an entry for each byte address, then for 32b addresses size of the table is
 - $2^{32} \times 32\text{b}$, i.e., 16GB
 - Entry per each word address: 4GB
 - Entry for each 256B unit address: 64MB

We saw for example, that if we were to associate, a piece of address translation information, with each byte address, so if the virtual address in the diagram here, we have an example of as address translation table. For each, in this example, for each virtual byte address, it has the corresponding physical main memory byte address, and the size of this table can be calculated. For example, if I assume that the size of an address is 32 bits, then I know that the width of the table, in other words the size of each physical address could be 32 bits, and I know that the number of entries in the table is going to be the number of bytes that can be addressed. That is going to be 2 to the power 32 and the size of the table would be an enormous 16 gigabytes.

So, the address translation table itself, the information which is needed to translate a virtual address to physical address would be much larger than most main memories. So, this is not clearly possible. We therefore, looked at what would happen if you did not

maintain a piece of address translation information, for each byte address, but rather let us say for each 256 byte unit of memory. In other words for the whole first 256 bytes of memory, in other words from the byte address 0 to the byte address 255; I might just have one address translation information inside the address translation table. And if we do this, then the size of the table comes down to a more manageable 64 mega bytes.

(Refer Slide Time: 04:40)



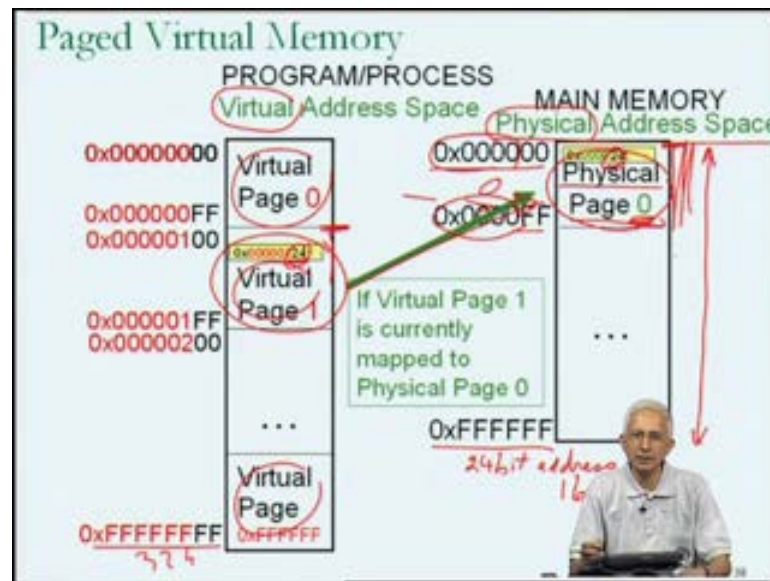
The slide is titled "Main Memory Management" in a green serif font. It contains a list of five bullet points, each starting with a yellow square. The first bullet point is "Example Paged Virtual Memory", where "Paged" is circled in red. The second is "Recall: Need for Address Translation". The third is "To translate a virtual address to the corresponding physical address, a table of translation information is needed". The fourth is "The size of the address translation table .." followed by a sub-bullet "□ ... can be reduced by not managing translations on byte basis but at some larger granularity". The fifth is "Page: fixed size unit of memory (contiguous memory locations) for which a single piece of translation information is maintained". A small number "39" is in the bottom right corner.

So, this is generally the kind of idea we are going to look at in the case of Paged virtual memory. Since the size of the address translation table is the problem, it looks like a possible solution is by not maintaining the address translations on a byte by byte basis but rather at some larger granularity, such as the 256 byte granularity, which I had suggested.

Now, technically the word, 'Page', is used for this size of granularity. So, it is the fixed size unit of memory for which a single piece of address translation is maintained. As I said in our example, I had just gone through; I was assuming that the size of each page is 256 bytes. And as a consequence, 256 contiguous neighboring memory locations will all be associated with the same piece of address translation information. And in effect we are breaking this memory down into units of size 256, in order to solve the problem of huge address translation table size.

So, you will recall that I talked about paged virtual memory. We now understand where the first term is coming from. We already know that we are talking about virtual memory, because the addresses which a process uses are not real addresses, but artificial or virtual addresses. Hence, the name of paged virtual memory for this mechanism for setting things up using fixed size chunks of memory contiguous as a unit for address translation.

(Refer Slide Time: 06:02)



Now, to get some idea about how this works, I am going to do this first to start off with a just a pictorial representation. So, we know that when a program is in execution, we refer to it as a process, until it does a **fork** after which it could be more than one process. But as far as the single process is concerned, we know that it has many things in memory, which must be present in memory for it to execute. This includes, as we know its text, which is its instructions, its static data, its stack allocated data, and its dynamically allocated data in the heap.

So, all of this, put together is what we refer to as the 'virtual address space' of the process for the program in execution. So, until now, when we talked about programs we were looking at the virtual address space in this way. We also know that this virtual address space can be viewed as the addresses from 0 up to some maximum possible address. So, in this notation rather than writing address 0 at the top and address two power 32 minus one at the bottom, I am showing you the addresses in hex notation.

Remember, whenever you see a number, which is prefix by 0 x; that means that it is being shown to you as hexadecimal or hex notation; hex is a short form for hexadecimal or base sixteen notation (Refer Slide Time: 07:17).

So, the address space here, you notice they are eight zeros in hex, which means that this is a 32 bit virtual address. So, **the minimum possible**, the first 32 bit address is all zeros and the last 32 bit Address is all Fs. And if I had expanded this into binary, it would have been all ones, which is the value 2 power 32 minus one. So, this is the way, that I would use a virtual address space in terms of its address space. Now, if I am going to try to manage the address translation information in units of 256 bytes, then the first 256 bytes would be at the top of the address space, going from byte address 0 up to byte Address 255. The second chunk of virtual address space of size 256 bytes would follow and so on. In fact, the actual addresses associated with each of these chunks, could be calculated. For example, I know that in the first chunk of size 256 bytes, I know that the first address is 0, which means that the last address is going to be 255. Because all the byte between 0 and 255 would constitute 256 bytes of memory, and if I convert 255 into binary and then into hexadecimal, I will find out that it is F F, six zeroes followed by F F.

Similarly, the second 256 byte chunk in memory, will start from 1 0 0, which is the address after 0 F F. The address in hex after 0 F F, is the address 1 0 0. And the second chunk would run from one 0 0 up to 1 F F and so on. So, I could be given this 256 byte granularity assumption, I actually can redraw the picture of the virtual address space, along these lines, chunks of size 256, with clear knowledge of what the first address in the chunk is, and what the last address in the chunk is. Now, I am going to be using the word page to refer to this chunks.

So, technically I could talk about the first chunk of size 256 bytes, which runs from address 0 up to address 255, or in hex, from address 0 0 up to address F F, based on the information, which I see in this diagram to the left. You know this that I have highlighted among the addresses of the bytes which are in that first chunk; I have highlighted what they have in common. And you noticed what they have in common is there all of those bytes, all of those byte addresses, the first six hex digits are all zeros. In fact, if I look at the second chunk of size 256 in the virtual address space, I notice that for it too there is commonality.

All of its bytes, what they have in common is that their first six hex digits are 0 1 or 1. If I was to convert that 0 0 0 0 0 1 into decimal, I would refer to it as one. Similarly, the 0 0 0 0 0 0 would refer to a zero. So, what that first 256 bytes have in common, is that the most significant bits amount to 0 and what the second chunk of 256 bytes have been common is that their most significant bytes, have in common is the one.

And therefore, ensure what I could do is to refer to the first 256 bytes in memory, since I am going to refer them as a page, and I know that this is the virtual address space, I can refer to those first 256 bytes as the virtual page zero, in other words, a virtual page in which all of the bytes have 0 as their common most significant bits.

Similarly, the virtual page one, as a virtual page in which all of the bytes have in common, the most significant address bits being one and so on. Another way to look at it is if I number the virtual pages from the first one in memory onwards, the first one would be called virtual page 0, the second one would be called virtual page one. But this is an additional observation of some value, that within virtual page 0, all the bytes have the most significant bits identical, as far as the address is concerned. And the commonality is the most significant bits amount to 0, which is the virtual page number.

In short, we have this picture (Refer Slide Time: 11:25) rather than thinking of the virtual address space of the process as text, data, stack, and heap, we can abstract out what is being stored at which part of the virtual address space and view the virtual address space as this sequence of virtual pages, starting with virtual page 0 and going all the way up to in this case virtual page hex F F F F F F, whatever that may be. We need not converted into decimal, but we will just leave it in that form.

Now, let us look at this little bit more critically. This is how we are going to chunk, the different bytes in memory, for the purpose of address translation. But, let us consider of one of these bytes in particular. The byte which I have highlighted happens to be in virtual page one, of the address space of the process that we are looking at. It happens to be in virtual page one, which is why the most significant bits of the address amount to 1. They are 0 0 0 0 0 1, and if I look at the least significant bits they contain the byte address 2 4.

So, if I was think about it carefully, within that virtual page one, the first byte would have had the least significant two bits 0 0, the second byte would have had one and so on. So, in some sense, I know that this particular byte that I am referring to (Refer Slide Time: 12:39) by address here in somewhere within the page, it is not the first byte, it is not the last byte, somewhere in between. And in fact, this 2 4 tells me exactly where it is located within the page. It is the byte just after the byte number 2 3, which is the byte just after byte number 2 2 and so on. So, I know its exact location within the page. So, by the page number, I know which page this byte is associated with, and the page number is given to me by the most significant bits of the address of the byte, and I within the page, I know exactly which particular byte is being referred to by looking at the least significant bits of the address. So, that is the generality idea of how these addresses are going to be viewed. Now, how is paged virtual memory going to manage an address space? Here, we have this picture of how it is going to view the virtual address space, but ultimately the objective is to translate virtual addresses into main memory addresses. So, into this picture we must include a picture which includes what main memory looks like. So, let me add that over here (Refer Slide Time: 13:41).

So, as far as main memory is concerned, what do we expect to see? We know that the address translation information is going to be managed in a table, and within that table that is going to be one entry for each virtual page, which means that they will not be a separate entry for each byte, but only one entry for each virtual page, which means that any one of these virtual pages can map into one unit in main memory, and the address of that unit in main memory, is what will be contained within the address translation table. Therefore, it follows immediately that main memory two must be viewed as being just a sequence of pages, and that is what we will refer to as the physical address space.

So, the physical address space is the range of main memory addresses, which actually exist. Hence, the name physical, it is a real thing, has supposed to virtual, which is artificial or imaginary. And for this particular example, I am going to assume that the size of the physical address space is such that the first address in hex is all zeros, and the last address in hex is F F F F F F. So, you will notice that this is a six hex digits, which means that this is a 24 bit address, which means that in actuality, I am able to unlike the virtual address space, which was 32 bit address, and which could therefore, refer to 4 gigabytes of information.

The physical address space is significantly smaller. That is the assumption, which is being made in this diagram and in fact, the size of the physical address space is only, if you do the calculation 16 megabytes. But, if that is all the memory that I have, the physical memory, the main memory that I have, then this would be the size of the physical address space. All the addresses from 0 up to what is the range of the address that are possible in the amount of main memory that I have.

So, this is clearly not an unreasonable example since you do know that they may be machines in which there are only 16 mega bytes of memory. As a consequence, the addressed space on that machine, in terms of physical addresses, would be only from hex 0 up to hex F F F F F F.

So, from a discussion a few seconds back, we understand that the physical address space must be viewed as being a sequence of 256 byte chunks, because there's going to be a mapping between a chunk here, and the chunk, a chunk from the virtual address space and a chunk on the physical address space. So, we therefore, view main memory to in this way. This is the first chunk of 256 bytes, which runs from address 0 up to address F F; physical Address F F, and then there is a second chunk and so on.

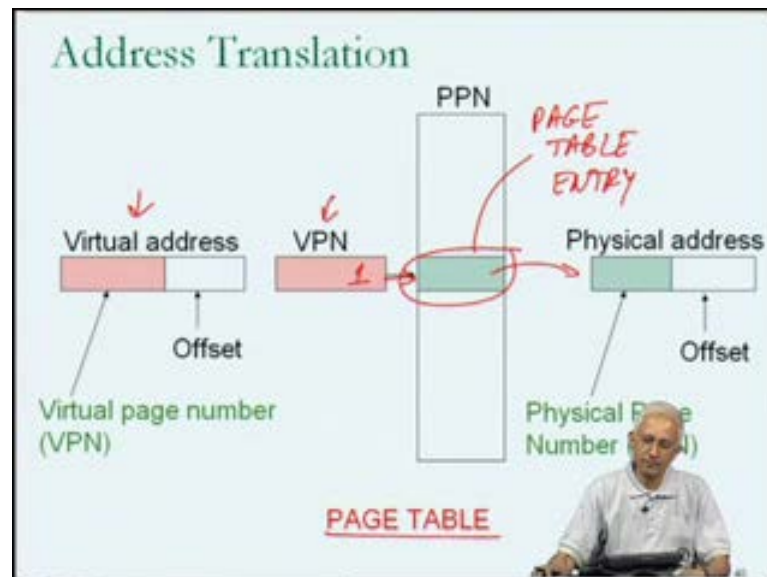
And just as we call the first chunk in the virtual address space, virtual page 0, we will refer to the first chunk in the physical address space, as physical page 0, and once again if I looked at the most significant bits of the addresses of all the bytes, in that first page, I would find out that they had the value 0, which is the virtual physical page number.

So, just like the virtual address space is a sequence of the virtual pages. The physical address page is the sequence of physical pages numbered from 0 on upwards and at any given point in time, the way things are managed by the virtual memory system is that a page in virtual memory will be mapped into a page in physical memory. So, let us suppose for example, that virtual page one, which is the page we are currently concentrating on, is mapped physical page 0, which is the page that we have in this diagram over here (Refer Slide Time: 17:11). What does this mean, it means that the operating system or inside the address translation table, there is a mapping information, which tells me that all the bytes which are associated the virtual page 0, or actually present in this range of the main memory.

That's what the address translation table will tell me, and more specifically if I was concerned about that yellow byte, the byte with address physical virtual address 0 0 0 0 1 2 4, then its position in main memory would be at the same displacement from the beginning of the page, as far as virtual page one was concerned. The same displacement from the beginning of physical page 0, and that information is entirely available to me by the number 2 4, hex 2 4. Therefore, the address translation table tells me where in main memory this particular page is going to be, the fact the virtual page is one is mapped to physical page 0, and the 2 4, in other words the byte address within that virtual page tells me specifically which byte in main memory this virtual address corresponds to, and that is the complete picture as far as the address translation is concerned.

This is basically what the hardware has to do, whenever there is a load or store instruction, in order to translate the address from a virtual address into an actual main memory or physical Address. Note that the address that we are talking about to the right or the actual addresses of the memory chips inside the computer, the thing which we called the main memory. These are the physically occurring memory cells within the computer.

(Refer Slide Time: 18:46)



So, what then is involved in address translation? So, the purpose of address translation is to translate a virtual address, as generated by the processor, into a physical address and you will remember that the processor generates a virtual address, whenever there is a in

the case of the MIPS one instruction set, whenever there is a load instruction, or a store instruction, or do not forget, whenever an instruction itself has to be fetched from memory.

So for any kind of data access out of memory or for the fetching of the instruction from memory, in all these situations, processor generates a virtual address. This virtual address must be translated before it can be interpreted or sent to memory for accessing the data or the instruction. So, we now look at the virtual address as the most significant bytes and those least significant bytes. For example, in the case of the particular byte that we were interested in, we have looked at the most significant bits of the address, which were 0 0 0 0 1, and the less significant which were that 2 4. So, the virtual page number, which was 1 for our particular example, I am showing in this red hashed region and the remaining bits of the address, I am showing in the un-hashed region.

So, what has to be done in address translation? We understood that as far as address translation is concerned, the net effect is going to be, that the 2 4 is going to survive, as the information about where within the physical page the particular byte that is being access is located. All that is going to change during address translation is that the most significant bits of the address are going to be change from in our particular example, from 1 to physical page 0 and that is what the address translation information would have contained.

So, the terminology that we will use is we now understand that when I look at a virtual address, I can look specifically at the least significant bits, which are just the offset within the virtual page and the most significant bits of the address, which are what we will now call the virtual page number; such as virtual page one or virtual page 0, all the way up to possibly virtual page in this case F's. And similarly, the most significant bits of the physical address of what we will now call the physical page number, I will abbreviate virtual page number as VPN and physical page number as PPN, and in passing again just note that the least significant bits of the address will not be affected by address translation. The 2 4, hex 2 4, in our example was even after translation was still hex 2 4, since it is only information about where within the page the information that is being accessed is located.

In essence, the address translation table contains this mapping from virtual page numbers to physical page numbers. This is what it might look like. The virtual page number is the pink bits, the most significant bits of the virtual address. The physical page numbers are those green bits, the most significant bits of the translated or physical main memory address. So, this is what we expect or you might see if you looked into address translation table. Now, we can do a little bit better than this, in the sense, that you notice that the virtual page numbers; the possible page numbers vary from 0 all the way to the maximum possible virtual page number, all F's.

If I was to structure the table along the lines of what is shown in this diagram then, in order to look into this table for a particular virtual address, I would have to search through the whole table. Unless, I maintain the table in order of virtual address virtual page number 0 at the top virtual page number one after that, virtual page number two after that and so on. In other words, if I structure the tables so that the first piece of transfer translation information in the table is the translation information for virtual page 0 or virtual page number 0 and the second piece of translation information is the translation information for virtual page one and so on.

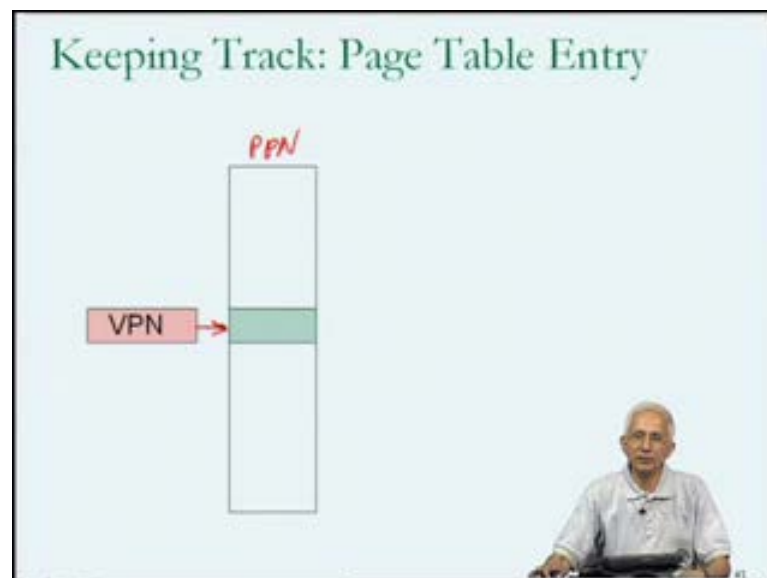
Then instead of having to search through the whole table, I can just index into the table using the virtual page number of interest to me. And if I was to structure the translation table in this way then there actually be no need to store the virtual page numbers at all. All that I would have to store would be the physical page numbers. I would then index into the page table or the address translation table, which contains all the physical page numbers using the virtual page number of interest to me. For example, since I am currently interested in translating virtual page number one, I would just look at, I would not look at the first entry in the physical in the address translation table, but I would look at the second entry which corresponds to virtual page one.

So, along these lines if I re reorganize the table; the table would look like this. So, the table contains only physical page numbers, and the way that you look into the table is using the virtual page number, which is what I am indicating over here. So, in order to translate this particular virtual address, I used the most significant bits of the virtual address, which of what we call the virtual page number to index into the page address translation table. And what I find there is the translation information, which is the actual physical page number, corresponding to this particular virtual page. From henceforth,

instead of talking of this translation table, referring to this translation table, as an address translation table, since we have organized in terms of these pages, I will refer to it as the page table. This is of a standard terminology.

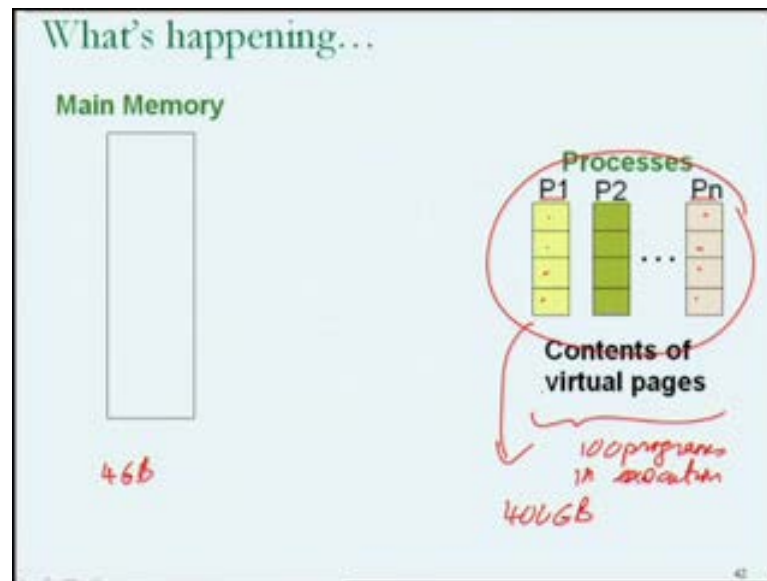
So, the page table is a data structure or address translation table of this kind, which contains one piece of address translation information for each page, for each virtual page of the address page under translation. And I could therefore, talk about any one of these entries in the page table as being a page table entry. So, this happens to be the page table entry for virtual page number one.

(Refer Slide Time: 25:08)



So, as we progress with our discussion of how page virtual memory is going to work, we will be adding more and more information into the page table entry. So, in order to keep track of this, I will, every now and then, augment this particular slide. So, just remember, at this point in time, what we expect is that the page table will contain one entry, for each virtual page and that the information contained in that virtual page in the page table entry is the physical page number for the page in question, in PPN. At this point, all that seems to be necessary, but let us go ahead.

(Refer Slide Time: 25:49)



At this point we have got an idea that in order to manage the size of the page table; it makes sense to maintain one page table entry, for each page rather than for each byte. If we had maintained one entry for each byte then we would not have called the page table, but address translation table of bytes. But, naturally, this does not really tell us how many processes are going to be able to share memory, share the same main memory.

So, we need to get the better understanding of what is happening. Once again, I will introduce the general idea pictorially. We will start looking at details. At this point we know that there is main memory. Main memory is a real thing. There are also the different virtual address spaces of the different processes, but at this point, I would not draw them, since they do not really exist in this diagram. I am not going to draw things which do not actually exist on the computer system. We just are going to have things which physically exist.

So, one thing which certainly does exist physically is main memory. Now, one thing which does not exist is the virtual address space or the contents of the virtual address spaces of all the processes that could be running on the computer system. Now, in this particular diagram, I am assuming that there are some n processes. There could be 10 processes, in which case last one would be p of 10 and numbering the processors from P 1 to P n . Each of these processes has its text, data, stack, heap, divided over a large number of pages, hundreds may be thousands of pages, and the note the terminology that

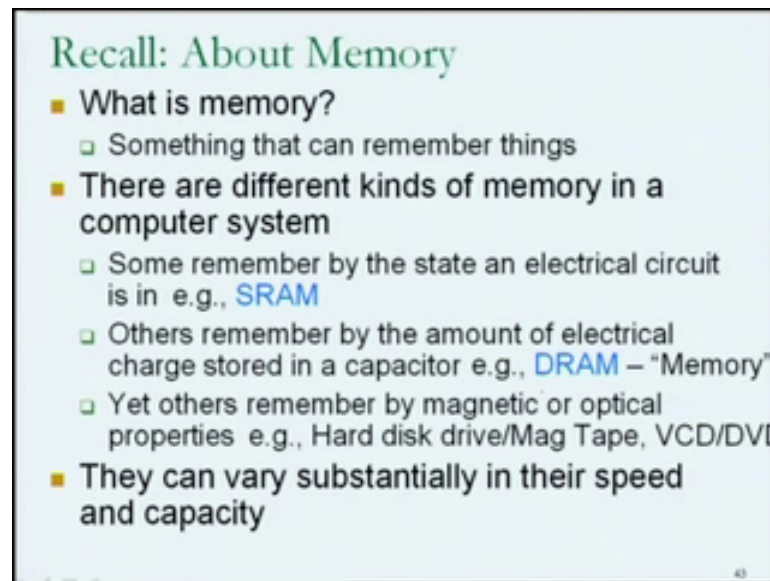
we are using is that these are, for example, these are the virtual pages of process one, these are the virtual pages of process n and so on.

Now, these are the contents of these virtual pages do not actually exist, **because they much**, I mean, as we saw, before they can be present in main memory. They have to be translated and it is conceivable that the size of main memory is much smaller than the sum of the sizes of all these virtual address spaces. Therefore, that is why the word virtual or imaginary is important here. These are things, which cannot be really exist in main memory for practical reasons. But, then the question would arise where the contents of these virtual pages can be stored.

It is not enough to just say that they cannot be stored in main memory, because there is not enough space. But, if I do require, let us say one hundred programs in execution to be supported. Let us suppose n is equal to 100. Then, I cannot just say yes, you can have up to 100 programs in execution. I must make it possible for the contents of the virtual pages of all those 100 programs in execution to be remembered. They have to be remembered somewhere in the computer system. They cannot just be or one cannot just say that they are fictitious they do not exist.

There must be some places for them to be stored, which raises the question of where can they be stored. Clearly, they cannot be stored in main memory. If you do a quick calculation, if I have hundred processes and each of them has 4 gigabytes of virtual pages, then they come to 400 gigabytes of virtual address space. Sum total of all these virtual address space is might be equal to 400 gigabytes and I might have 4 gigabytes of main memory. So, the main memory is not the answer to this question of wherever the contents of these virtual address spaces pages be stored.

(Refer Slide Time: 29:13)



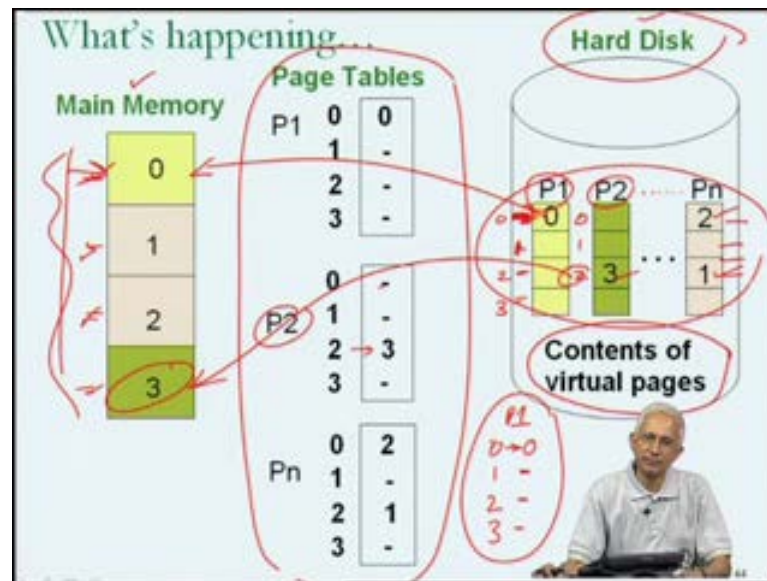
Recall: About Memory

- What is memory?
 - Something that can remember things
- There are different kinds of memory in a computer system
 - Some remember by the state an electrical circuit is in e.g., **SRAM**
 - Others remember by the amount of electrical charge stored in a capacitor e.g., **DRAM** – “Memory”
 - Yet others remember by magnetic or optical properties e.g., Hard disk drive/Mag Tape, VCD/DVD
- They can vary substantially in their speed and capacity

So, we have to look back at the other kinds of memory that are present in a computer system. And you will recall that when we talked about memory for the first time, I said, “we use term memory to refer to anything they can be use to remember things”, and we are currently in a situation where we have to find some form of memory, which can store a large amount of information; 400 gigabytes of information contained in all the address spaces of all those 100 programs in execution. And very clearly memory, main memory is not the answer to that question. We have to be looking at something, which has much larger capacities, such as potentially hard disk drive.

Now, we can rule out the possibility of V C D, D V D. Since, very clearly, we will need some kind of memory, which can be modified and V C D, D V D are not that easy to modify. You do have rewritable D V D's, rewritable V C D's. But they cannot be really rewritten in a whole a large number of times and may not be that useful in this scenario. Magnetic tapes are not that widely available on computer systems today, which leaves us with only the option of using the hard disk drive, to store the virtual address spaces of all the programs in execution.

(Refer Slide Time: 30:30)



And therefore, that is what I will do. Will assume that in drawing this diagram, which is telling us what is happening behind the scenes. When addresses get translated, I must include the hard disk drive. So, I will do that. So, in a sense I am going to include the hard disk drive. I am showing the hard disk drive by this cylinder. It is a just as a notation. So, this is the hard disk drive and the contents of the virtual pages of all the programs in execution, all the processes for the moment, I am going to assume as stored in the hard disk. So, they are been remembered and that was our requirement.

So, a large number of these pages are occupying the hard disk drive. And as I said for 100 processes, this could be as much as 400 gigabytes of information, which is being stored on the hard disk drive. Now, at any given point in time, only some of those pages can be present inside the main memory. We have seen that the main memory is much smaller. Main memory may only be size 4 gigabytes and therefore, very clearly all those 400 gigabytes of pages cannot be stored in main memory.

So, some subset of those virtual pages of all the processes, which are in memory, will be actually available in the main memory of the computer system. Just note that I am not talking about virtual address, spaces anymore. I am talking about the hard disk, which is a real part of the computer system. I am talking about the main memory, which is a real part of the computer system.

So, the fact of the matrix, as I said some subset of these virtual pages which are in available in terms of the content on the hard disk. So, some subset of them will be present in the main memory. Let us suppose for the moment that for simplicity, I am in this diagram. I am showing you a virtual; each of these processes has a virtual address space made up of on the 4 pages.

This is a (()) simplification, but it will make the diagram a little bit easier. So, there only 4 pages in process P 1, each of the processes and in a similar simplification, I am assuming that there are only 4 physical pages in the main memory. And once we understand this example, which has only 4 pages in each virtual address space and only 4 pages in the physical address space. It will be easy to extend the understanding to more realistic scenarios.

So, the current assumption that I am making is, since main memory has space to remember only 4 pages, given the size of main memory, only 4 out of all these virtual pages can be present in main memory, at given point in time. And in the current situation, it looks like the first page of process P 1 is present in main memory, along with two of the pages of process P n and one of the pages of process P 2. So, those are the 4 selective pages, which are currently present in the main memory. All the other pages are not present in main memory right now, but are safely available on the disk. What then will the address translation tables or the page tables for each of the processes look like? We can actually construct the address translation table for process P 0 quite readily.

Now, the address translation table for process p 0, I am sorry. I am referring to it is process P one. We will have to contain one piece of address translation information for each of its four pages If I refer to its 4 pages as 0 1 2 and 3. Here, I am trying to figure out what the address translation table or the page table of process P 1 will look like.

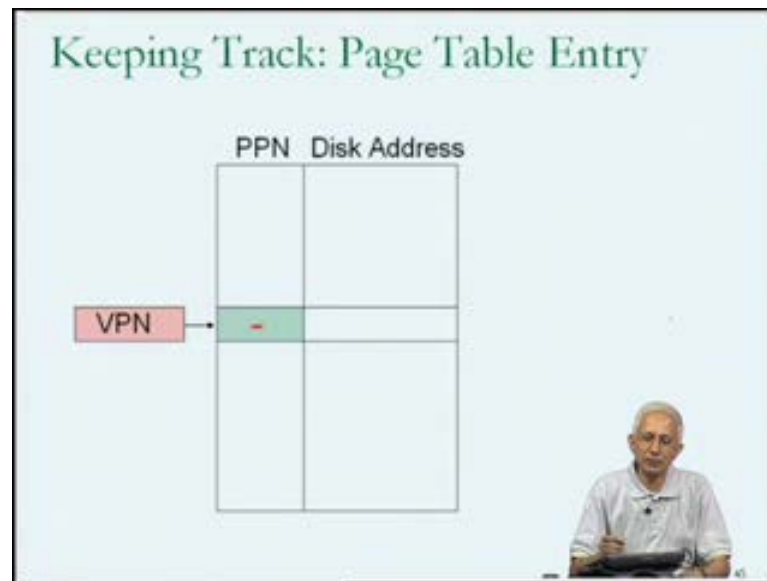
So, what is the address translation information associated with its page 0? I know that its page 0, the one which is at the beginning of its address space, is currently present in physical page 0. Therefore, if I looked at the page table, I would find that 0 maps to 0. What about second page of page one, in other words, the page whose virtual page number is one? Remember, the virtual pages of page of this process of virtual page 0, virtual page 1, virtual page 2, and virtual page 3, and it has only four virtual pages. Virtual page 0 is currently mapped into physical page 0. Virtual page 1, as we can see

from the left hand side, is not present. And therefore, that would have to be indicated in the page table entry. There is no mapping for virtual page one. Similarly, there is no mapping for virtual page 2 and there is no mapping for virtual page 3.

So, this is in fact, what the page table of process one would look like. Similarly, I can work out what the tables of process 2 and process n would look like. So, just to confirm, as far as process 2 is concerned, its virtual page 0 is not present in main memory. Its virtual page 1 is not present in main memory. Its virtual page 2 is present in main memory, in the third physical page. Hence, when I look at the page table entry of process two, I find out that virtual page 0 not present, virtual page 1 has no mapping; virtual page 2 is present in physical page 3. There is a mapping from virtual page 2 to physical page 3 and similarly the page table entries for all of the processes within the computer system.

So, this is the collection of all page tables for this particular example. I am not showing you the remaining processes, but they would have similar entries. As you could imagine if they are 100 processes given that the size of main memory of the page table entries for all the other processes would have no meaning full mappings. They would just have dashes for all their entries. So, this is the address translation information and this also tells us that this big step that we took where we said that the contents of all the virtual pages of all the processes will be present on hard disk, gives us a clear indication that we will have to keep track of the addresses on the hard disk of each of the virtual pages of each of the processes.

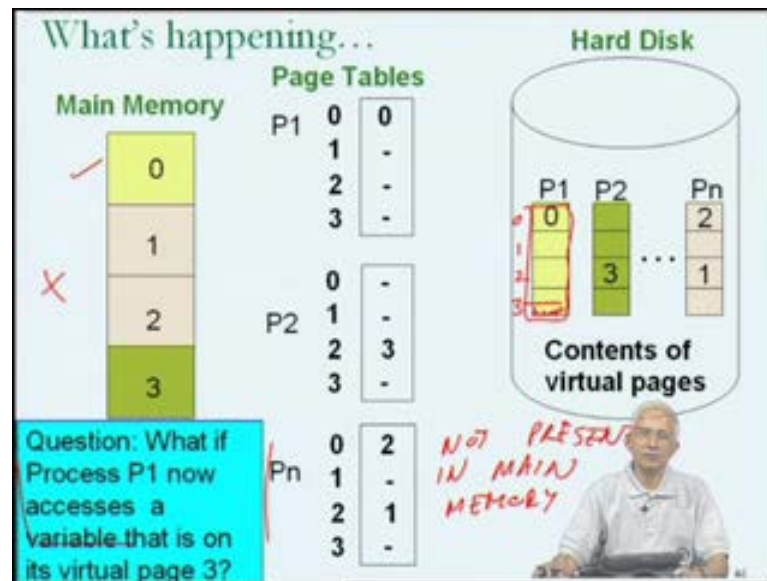
(Refer Slide Time: 36:42)



This means that if I go back to my picture of what a page, the information that a page table entry must contain, I will clearly have to add to that. Look at this way. If there is a virtual page, which is currently mapped to physical page then they will actual be an entry inside its physical page number. But, if there is a virtual page, which is not mapped to a physical page, then that means it is not present in main memory. But, clearly one must keep track of where it is present on the disk, for which reason, we will add one more entry into the page table entry; into each of the page Table entries and that is basically the disk address of that particular virtual page. Just note, we have been assuming that copies of all the virtual pages are present on the disk. Since, they are going to be thousands or millions of such virtual pages present on the disk you will have to know the address of each of them, in order to read it out of the disk as in when it is needed.

Which is why, I add an entry, a field into each page table entry, which I will refer to as the disk address. And for the moment we do not know what a disk address would look like. We just assume that it is some information using which that particular page can be read out of the disk and we learn more about disks later. But, for the movement you could assume that the disk addresses 30 or 40 bits of the information using which the page can be identified located on the disk.

(Refer Slide Time: 38:12)



So, that is how we end up with this picture over here, and the situation as we have it right now is clear for 4 pages, if the address, if the virtual address space of each processes of size 4 pages. What if it is much bigger, then this going to be a lot more information stored on the disk, the size of each page table entry is going to be larger and hopefully the size of main memory will be larger than what we have over here, this simple example of four pages. Now, this raises an interesting question.

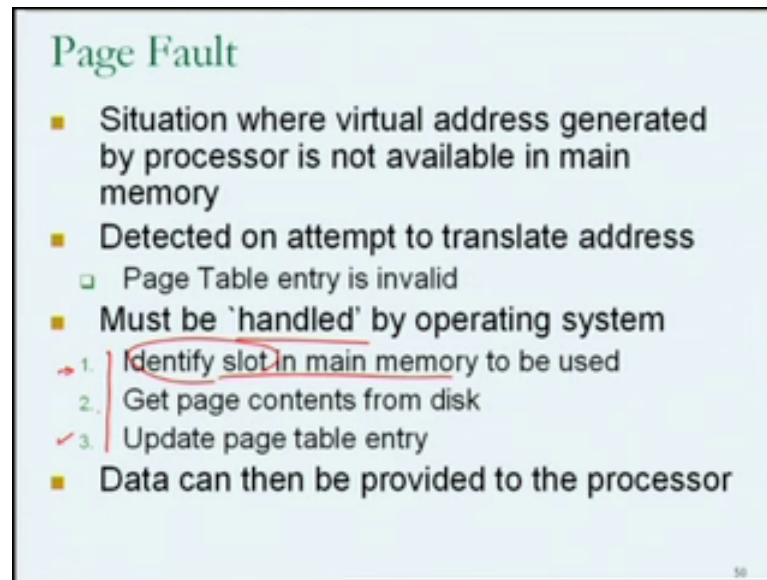
So, the question is this. "What if the system is in this particular state?", and process P 1 now accesses of variable that is present on its virtual page 3. Now, what is virtual page 3 of process P 1? Remember, virtual page 0 is present in main memory. Virtual page 1 is not, virtual page 2 is not and virtual page 3 also is not.

So, again just to make sure we understand what is being said. The question is what if process P 1 is in execution and it accesses the variable that is in its virtual page 3. What could be present in its virtual page 3? Remember that this is the virtual address space of a process, so it is possible that if the heap of the process is growing from bottom upwards that it could be a dynamically allocated variable that we are referring to over here.

So, one of the variables of the process is accessed by the program in execution. A load word instruction is executed on that address. That is a scenario that is being raised over here. Now, the problem as you can see it is the that particular page virtual page of

process P 1 is not present in main memory, and therefore, cannot be accessed using a load instruction, which the load instruction can only be used to actually access something, which is in main memory. If it is not in main memory then clearly it is not an achievable, it is not accessible unless some more work is done behind the scenes.

(Refer Slide Time: 40:35)



Page Fault

- Situation where virtual address generated by processor is not available in main memory
- Detected on attempt to translate address
 - Page Table entry is invalid
- Must be 'handled' by operating system
- 1. Identify slot in main memory to be used
- 2. Get page contents from disk
- ✓ 3. Update page table entry
- Data can then be provided to the processor

Now, this situation that I have just been described, one way a process tries to access a variable and it is identified that that particular page is not present in main memory. This situation is what is referred to as a page fault. So, this is a technical term. This is the situation where a virtual address generated by the processor is translated or in attempt is made to translate it, and it is found that this particular virtual entity is not currently present in the main memory.

So, the question is what can be done. Clearly, we cannot say the bad luck of the programme, so program will have to stop. Very clearly, the program must be allowed to continue execution. All of these features about address translation were added. So, that many programs could correctly execute sharing the resources of a computer system. A page fault is hopefully just a minor tumbling block towards the program is being able to continue execution. But, clearly also the page fault must be rectified. Something must be done to remedy the situation. As a result of which, the piece of data, which the program is looking to access should become available in main memory.

Now, the page fault would have been detected when an attempt was made to translate the address and as you will notice from the previous slide (Refer Slide Time: 41:37) and this particular example when process P 1, address or something on page 3, when an attempt was made to translate the address, it would have been noticed that there is no mapping, which is the clear indication that there is a page fault. The page required, the virtual page required.

(No audio from 41:54 to 45:51)

the page fault is what we will refer to as the page fault handled shortly.

Now, what must the page fault handler do now? In order for the processor's request, for that particular piece of information, to be satisfied the page containing that piece of information, must be brought into the main memory, from its location on the disk. In order for it to be brought into the main memory, one particular slot of one particular physical page, in the main memory must be identified for use, by the page that is now going to be brought in from the disk.

So, that is the first step, as far as handling the page fault is concerned. You must identify a slot or a physical page number to be used in main memory for this particular page. After this the page could conceivably be read from the disk. Remember that each virtual page has its presence in disk as a copy.

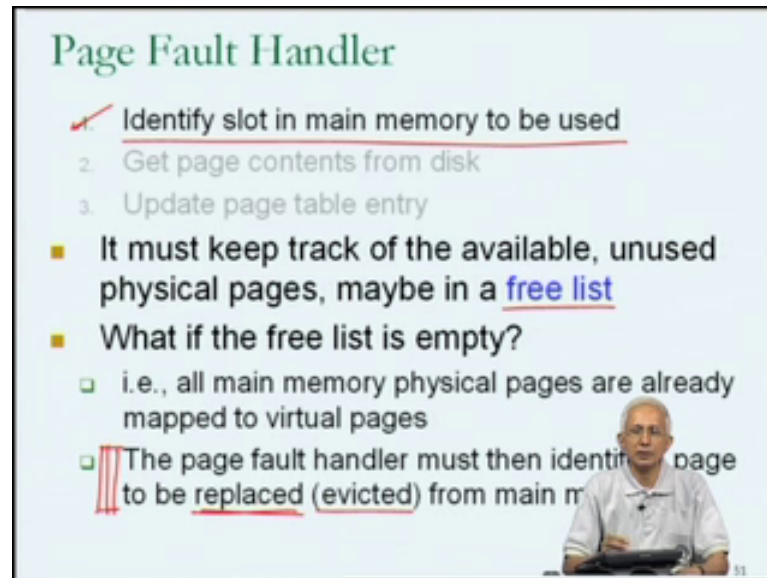
So, the disk read can be used to read the contents of that page from disk into the slot or the physical page in main memory. After this, the page table entry can be updated, can be updated in setting. It is valid to be one and putting the correct identity of the page physical page number into its PPN field. So, that is all that we mean by updating the page table entry.

So, this is essentially what has to be done by the operating system's page fault handler. So, the page fault handler is going to be a small piece of code inside the operating system to handle this possible scenario, the scenario of the page fault.

So, once this has been done, the load word instruction can be successfully executed by the processor. So, the processor which had executed or try to execute load word instruction, as a result of which in address translation was attempted, as result of which

the page fault was identified, which initiated the execution of the page fault handler, can now return back and complete execution of that load word instruction. That is what it boils down to.

(Refer Slide Time: 47:54)



Page Fault Handler

1. Identify slot in main memory to be used
2. Get page contents from disk
3. Update page table entry

- It must keep track of the available, unused physical pages, maybe in a free list
- What if the free list is empty?
 - i.e., all main memory physical pages are already mapped to virtual pages
 - The page fault handler must then identify a page to be replaced (evicted) from main memory

51

Now, there is one problem, which arises in this context and that is it may not be as easy as you would think to identify a slot in main memory, to be used in order to bring in a copy of the virtual page, required by the process in question and let us see why.

Now, from the suggestion that free slot in main memory is to be identified, the immediate suggestion is that the operating system must keep track of the available unused physical pages. If it keeps track of which pages in main memory are currently not used then identifying the slot in main memory is quite easy. It will just take one of those pages from among all the physical pages, which are currently not being used for any virtual page. We can just pick one and you could maintain all.

The operating system might maintain all this information in a single linked list. For example, this might be called a free list. Therefore, if that is, if the situation is that the list contains information about free physical pages, then step one of the page fault handled is quite easy. Just take the entry from the front of the free list and use that particular page as the slot in main memory where the virtual page can be read from disk. But what if the free list is empty, as in the example that we had in our diagram where all

of the four physical pages in main memory are currently being used and therefore, the free list of unused pages in physical memory would be empty.

What if the free list is empty once again, this cannot be used as an excuse to terminate execution of the program, which causes the page fault. This two must be correctly handled. **This must I am sorry**. This must be covered by the page fault handler. This is just another contingency. They have to be taken into account. So, what should be done if the free list is empty? In other words, what if all the main memory physical pages are already mapped to virtual pages as in our four page example?

Then what the page fault handler will have to do is to actually make space within the main memory, by evicting one of the currently used main memory pages or physical pages. So, the page fault handler must then identify a page to be replaced or evicted from main memory. Once, one of the currently used pages in main memory is evicted or eliminated, and then the required page, which causes the page fault, can be read from disk into that particular location in the main memory.

So, this is an additional functionality that would have to be taken up by the page fault handler. The task **curve** identifying a page for replacement or eviction from main memory and what I mean by eviction or replacement is actually removing that page from main memory, in order to create an empty page in main memory, which can be used for some other purpose. And in this particular example, the other purpose is providing space for an unused page, which is available only on disk to be stored in main memory, so that the process which accessed as variable on that page can successfully continue execution. So, we have to reach the point where we need to understand on what principles an operating system page fault handler could decide which page it can replace from main memory, in order to make space for another virtual page. And as you can see, we are coming to a point, in our discussion of main memory where, which may be of interest to us, as far as people who write programs are concerned.

Because with this understanding, we realize that there is a possibility that when our program is in execution it may suffer situation where some of the pages which it is using in main memory may actually get replaced in favor of the pages of some other process. And if this is the case and we do need to know a little bit more about how that decision about which page to replace is made by the operating systems page fault handler.

So, this is a somewhat important consideration, which will have some impact on the efficiency with which our programs execute. And since this is such an important break point, I will stop here for today. When we continue in our lecture number fourteen, we will try to understand the kinds of considerations which are used in the operating system page fault handled in order to decide which page can be replaced from memory, from among all the pages, which are currently present in the physical memory, which one can be replaced or evicted or thrown out of a physical memory to make space for another page which is currently required.

Thank you.