

Computer Organization and Architecture: A Pedagogical Aspect

Prof. Jatindra Kr. Deka

Dr. Santosh Biswas

Dr. Arnab Sarkar

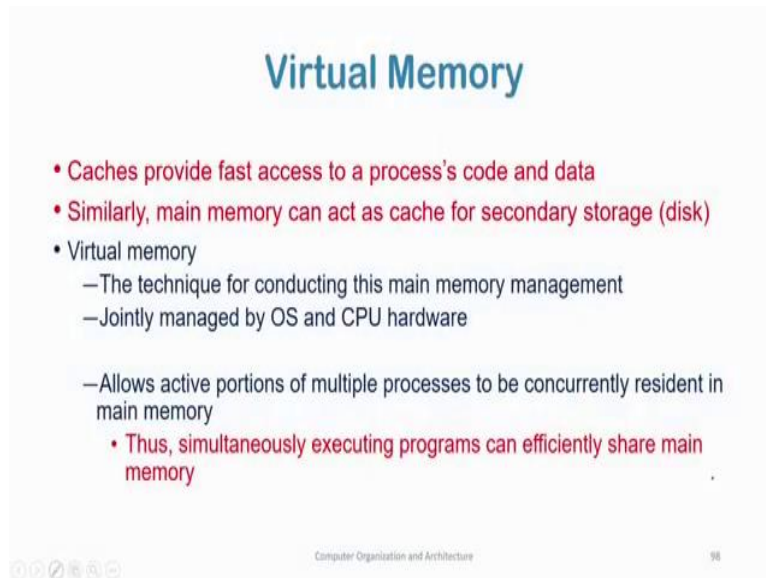
Department of Computer Science & Engineering

Indian Institute of Technology, Guwahati

Lecture - 27

Basics of Virtual Memory and Address Translation

(Refer Slide Time: 00:27)



Virtual Memory

- Caches provide fast access to a process's code and data
- Similarly, main memory can act as cache for secondary storage (disk)
- Virtual memory
 - The technique for conducting this main memory management
 - Jointly managed by OS and CPU hardware
 - Allows active portions of multiple processes to be concurrently resident in main memory
 - Thus, simultaneously executing programs can efficiently share main memory

Computer Organization and Architecture 98

In this module we will start our discussion with Virtual Memory. In the previous modules we had studied caches which provide fast access to a processes code and data. In a similar way we will now study a technique which allows the main memory to be used as a cache for the secondary storage. And this mechanism is jointly managed by both the OS, and the CPU hardware.

So, virtual memory allows active portions of multiple processes to be concurrently resident in the main memory. So, as we know that multiple processes must exists together in the main memory for being executed. So, when a program is being executed it must exist on in the main memory.

(Refer Slide Time: 01:32)

Virtual Memory - Basics

- Before execution, each process gets a range of virtual memory locations to address its data and code
 - Virtual or logical address space of the program
 - Range example: 0 to $2^{32} - 1$
 - The CPU generates virtual addresses
 - Addresses seen by the memory unit are physical addresses
 - CPU and OS together translates virtual addresses into physical addresses
 - This translation process
 - enforces protection of a program's physical address space from other programs
 - isolates OS from user processes
 - allows a single user program to exceed the size of main memory

Computer Organization and Architecture 99

Now virtual memory allows the main memory to be shared between multiple programs. Before execution each process gets a range of virtual memory locations to address its data and code. So, as we are talking this is virtual memory it does not exist in practice.

So, when we are combining when the program is being compiled it assumes that it has a range of memory locations at its disposal. So, this it is virtual or logical address space of the program and this can be as big as the entire addressable space of the processor; for example, in a 32 bit processor, where address buses are of length 32 bits. So, the range of memory addresses that the CPU can address goes from 0 to $2^{32} - 1$. So, each process in the in the in the virtual memory managed system can access as big a memory as $2^{32} - 1$. So, therefore, the each program or the process generates virtual addresses, now addresses seen by the memory unit or physical addresses.

So, the virtual addresses that are generated by the CPU must be mapped to physical addresses. So, I generate a virtual address there will be a mapping process, the memory management unit will map this virtual address into a physical address; that means, where this data that I have asked that I have accessed will actually reside in main memory.

So, that mapping will be done to that physical address and this physical address will be floated into the memory address register of the system, and from this memory address register this address will be floated onto the memory unit to access the data.

Now this translation from virtual addresses of virtual addresses of a process to the physical address is done together by the CPU, and OS. And what do you get additionally that this translation process also enforces protection of a programs physical address space from other programs.

So, now why this is possible we are saying that each process can address a virtual address space can provide its code the code and data corresponding to a process can be within its virtual address space will be within its virtual address space, this virtual address space will be of the size from 0 to $2^{32} - 1$ let us say. Let us assume for a 32 bit computer. And then this virtual address will be mapped to a physical address in main memory. Now this mapping process will be to valid locations in the physical memory where data and code corresponding to this program exists.

And this mapping process or this translation process from virtual to physical address does this mapping correctly that is that is this mapping from virtual address to physical address will be done only to those physical addresses which belong to this program. And it will not allow access to other portions of the physical memory where data and code corresponding to other programs reside.

So, this enforces a protection of a programs physical address space from other programs. So, why is this needed? Because we said that multiple programs together exist in main memory. Now one program should not encroach into the address space of another program, this protection is enforced by the translation process.

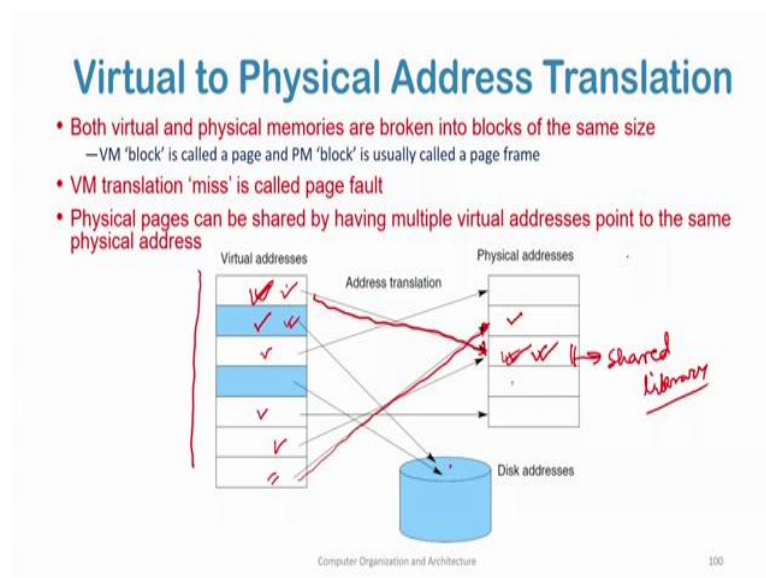
It also isolates the operating system which is which is a separate program from other user processes. Because the operating system is protected, it conducts and nicely executes all conducts all the processes happening in the entire system. So, that has to be protected from user programs, OS runs in kernel mode it has to be protected from user programs. And the translation process also ensures that user programs do not encroach into the code space of the OS in main memory.

Now, the translation process also allows a single user program to exceed the size of the main memory because virtual address space and physical address spaces do not have a one to one mapping meaning that virtual address spaces can be very big as big as the size of as I said 2^{32} in a 32 bit computer.

And let us say in my physical address in RAM is only say 256 MB, which is much smaller. Now the entire code the virtual entire code and data of the process will remain in the secondary storage. And portions of the portions which are active at a given time which is defined by something called a working set we will discuss later will only be resident those portions which are active at a given time if those portions of the program which are active at a given time will actually be resident in the main memory. So, the entire virtual entire code and data of the process need not be resident in the main memory together.

So, the virtual address space of a program can be much larger or arbitrarily large and has no relation with the size of the main memory. The virtual address space can be larger or smaller compared to the main memory that we have.

(Refer Slide Time: 07:24)



Now how do we do this mapping from virtual to physical addresses? So, both the virtual and physical both the virtual and physical memory; so the virtual memory and the physical memory are broken into blocks of the same size and this block. So the blocks are of the same size. As for caches we divide into blocks and lines for virtual memories. And physical memories we divide virtual memory into blocks called pages and the physical memory into blocks of the same size called page frames.

Now, the virtual memory translation miss is called a page fault; that means, during translation I have my CPU asked for data corresponding to a virtual address.

Now how do I get that data I do the mapping so that I find out the corresponding physical memory location in which this virtual data corresponding to this virtual address reside? Now when I am trying to do this mapping I am saying that the that the page the corresponding page in physical memory corresponding to the virtual page that I need is not there in physical memory currently.

Now that has to be brought from secondary memory or disk, now this constitutes a page fault when the page that I require the virtual page that I require is not there in main memory in a page frame of the main memory. I have a page fault here in this figure, we see that we have virtual addresses on one side these are the virtual addresses each of them is a virtual page these are pages.

Now the white pages, these I have a mapping for them in the physical memory. So, I have a mapping for them in the physical memory. So, corresponding to this page I have the content of this page is in the physical memory this page frame this for this virtual page I have the content in this page frame.

However for this page I do not have I do not have a corresponding mapping in the physical memory; that means, this page is currently not resident in a physical memory page frame. So, what happens this when I access this virtual page I have a page fault, and I have to bring this page from the disk to physical memory and only then can the data or the code that I need corresponding to this page can be accessed.

So, I have to service the page fault and only then can the data corresponding to the page which is not resident in main memory be accessed. So, and the other point is that pages can be shared by multiple virtual addresses.

So, the same physical memory page can be mapped can be pointed to by multiple virtual addresses for example, in this case here this virtual page and this virtual page points to the same physical page frame this one. Now when is this important so, there could be code or data which is accessed or shared by multiple programs say. For example, let us say we have a dynamic; dynamic linked libraries which I shared files in the system or let us say we have a language library say our printf program.

Now, all C programs may need to use this printf there are 2 options for this. Now each program can have a copy each process can include a copy of the printf program into its virtual address space will have a mapping of this printf program into its virtual address space.

So, program 1 C program 1, and C program 2 both accesses the same printf program. Now they can the so if it is the same printf program, it is better to share the code instead of each program having a separate copy of that printf program. So, therefore, this is done through a mechanism called dynamic binding.

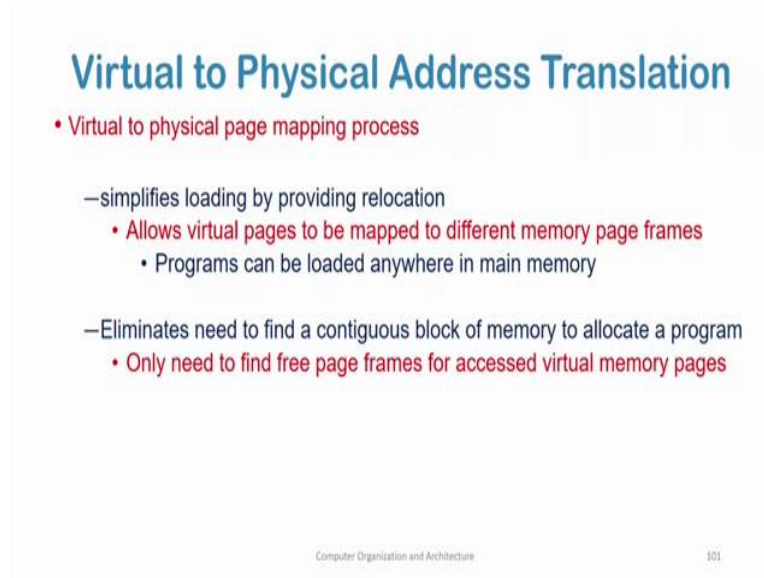
So, what happens is that for shared libraries you do not actually load the libraries as part of the as part of the address space in your in your executable code. Instead you have you just include a stub saying that printf is a shared library which I may need to access from a part of the program. Now, when this shared library is first accessed during execution I first check whether this shared library is already existent in main memory or not, if it is existent in main memory I just include another link to this shared library code instead of instead of having a separate copy of it.

So, although I bring for the first time I bring this shared library into the main memory when another program suppose tries to access the same shared library then what I will do I will not bring the copy from the secondary memory to storage I will not bring another copy of the print function of the shared library from secondary memory to storage. I will reuse the same code that I already have in main memory. I will just link I will just link from the other program that the printf code exists in this part of the main memory instead of instead of copying having another copy of the of the of this shared library.

So, how will I achieve this? I will achieve this by having multiple virtual pages to point to the same physical page frame. So, let us say this physical page frame here contains the code for the shared library, contains the code for the shared library.

So, now this shared library can be accessed by now and let us say this virtual page is corresponding to program 1, and this virtual page is corresponding to program 2. Therefore, when both these pages point to the same shared library code this can be done by what by having these 2 pages point to the same page frame here in main memory. So, physical pages therefore, can be shared by having multiple virtual addresses point to the same physical address, virtual to physical page mapping simplifies loading by providing relocation.

(Refer Slide Time: 15:06)



Virtual to Physical Address Translation

- Virtual to physical page mapping process
 - simplifies loading by providing relocation
 - Allows virtual pages to be mapped to different memory page frames
 - Programs can be loaded anywhere in main memory
 - Eliminates need to find a contiguous block of memory to allocate a program
 - Only need to find free page frames for accessed virtual memory pages

Computer Organization and Architecture 101

As we have seen in a virtually managed memory system the location in the virtual memory is with respect to the start of the virtual page location. Now these virtual pages can be located anywhere in physical memory.

So, therefore, if I have provided an address and I know the start of the page in main memory I can locate it. So, therefore, it simplifies loading of the program so the program can be loaded anywhere into the physical memory there is no restriction, the virtual pages need not even be contiguous.

So, the virtual pages have can be mapped by themselves in without relation to other pages at any portion of the physical memory. Logically speaking there are restrictions which we will not go to at this point in time; however, we can assume here that any virtual page can be mapped to any physical page frame. So, allows virtual pages to be mapped to different memory page frames. So, this is relocation.

So, in my virtual address space my code was contiguous one after another virtual page one followed by virtual page 2 followed by virtual page 3. However, virtual page 1 can reside in let us say page frame number 10, and virtual page 3 can reside in page frame number 1. There is no there is no issue that because this virtual page frame was because virtual page number 3 was after virtual page number through virtual page number 2 because virtual page number 3 was after virtual page number 2. It need not be that in the physical page frame also it will reside

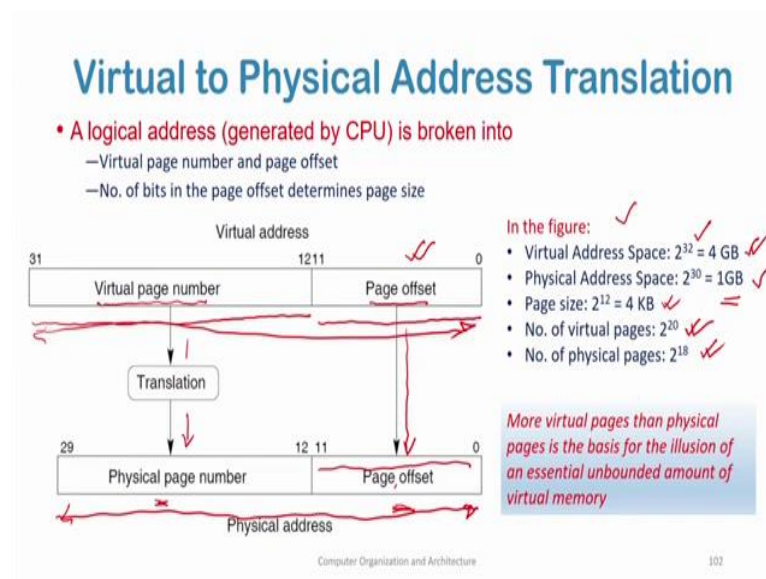
in physical page frame number 2, followed by physical page frame number 3, it need not be so.

So, virtual page number 3 can reside very well reside in physical page number 1, and virtual page number 2 can very well reside in physical page number 10 no problems. So, programs therefore, can be loaded anywhere in physical memory. And this translation also eliminates the need to find a contiguous block of memory to allocate a program.

So, if we did not have such a paged memory system where I have the same page size I have broken both the physical memory, and the virtual memory into pages of the same size. If I did not have a system like that then what would I have to do when I have to place a process in main memory I have to find out appropriate amount of contiguous block of memory for this process in main memory. Otherwise I cannot place this entire process in main memory.

Now, I need to only find whether I have a free I have free page frames for a given virtual page which is being accessed by the CPU I have a virtual page which is being accessed by the CPU. If it is currently not there in the main in a main memory page frame I just need to find or replace a page frame in the main memory find a find a given page frame corresponding to this virtual page which I need place my data from the secondary storage into this page frame, and then access the data after that.

(Refer Slide Time: 18:46)



So, virtual to physical address translation a logical address generated by the CPU is broken into a virtual page number and a page offset. So, here I said that my if in let us say I have a 32 bit address space, 32 bit address bus. So and I have a 32 bit virtual address space, so, my virtual addresses will be of length 32 bits in this figure we see that the virtual address or also called the logical address is broken into 2 parts the virtual page number and the page offset.

So, the higher order 31 to 12 these 20 bits I am using for the virtual page number and these 11 to 0, these 12 bits I am using for the page offset the number of bits in the page offset determines the page size. So, here because I have used 12 bits in the page offset the number the maximum number of bytes or words in my page can be 2^{12} ok. So, I have 4 KB pages in this organization now the page offset is translated as it is because the so this portion this part of the address tells me which byte within this page do I require and this is translated as it is in the main memory, this does not change.

The virtual page number this part now goes through a translation goes through a translation process and for that I generate a physical page number. So, my CPU has generated virtual address my CPU let us say my CPU has asked for a data from a virtual address which is given by these 32 bits. Then these 32 bits is broken into 2 parts the lower order 12 bits here forms the page offset. And it tells me which byte this address these bits the values of these bits tells me which byte within my page do I need to access at this current time. The higher order bits from the 12 to 31 these bits tell me the virtual page number.

So, how many page, how many virtual pages can I have? I can have 2^{20} pages. Because this 12 to 31 means I have 20 bits for the virtual page number, so the number of virtual pages can be 2^{20} . Now each of these virtual pages can potentially be translated to a physical page number. Now for this virtual page number I will have a corresponding physical page number, we see that the number of bits in this virtual address and this physical address are different. So, this virtual address has 20 bits.

However, the physical address we see it is from 12 to 29 so, it has 18 bits. So, I so, the number of physical memory page frames here is only 2^{18} . But the number of virtual pages that I can have for this process is 2^{20} ok. Now this one is the physical address so this whole thing will be floated after translation will be floated onto the memory address register. So, the page offset part these bits will be will be just placed as it is in the physical memory address.

This physical page number these bits will go through a translation and this 20 bit address will be transformed to a corresponding 18 bit address after the translation and after the translation we will together add up this physical page number and this page offset to get the entire physical page, physical address.

This is the frame address and this is the byte within the page frame that I require, this is the byte within the page from that I require and this is the frame address. So, in this figure the virtual address space is of 2^{32} . Because my I have 2^{32} different addressable locations in the virtual memory. The physical address space is of 2^{30} from 0 to 29.

So, the size of the physical memory is 1 GB, the size of the virtual memory is 4 GB, the page size is 2^{12} which is 4 KB, 2^{10} is 1 KB.

So, 10 bits 1 KB and 2 bits more so 4 KB's the number of virtual pages as we discussed is 2^{20} and the number of physical pages is 2^2 to the power a number of physical pages is 2^{18} . So, more virtual pages than physical pages is the basis of the illusion of an essentially unbounded amount of virtual memory.

Now the virtual memory can be as big as the address bus that I have. if I have a 64 bit address bus is in a 64 bit computer, then the number of the virtual address space can be as large as 2^{64} .

So, essentially I have unbounded amount of memory. However, the translation process guarantees that all these virtual pages, or virtual addresses will be correctly mapped to proper locations in the physical memory.