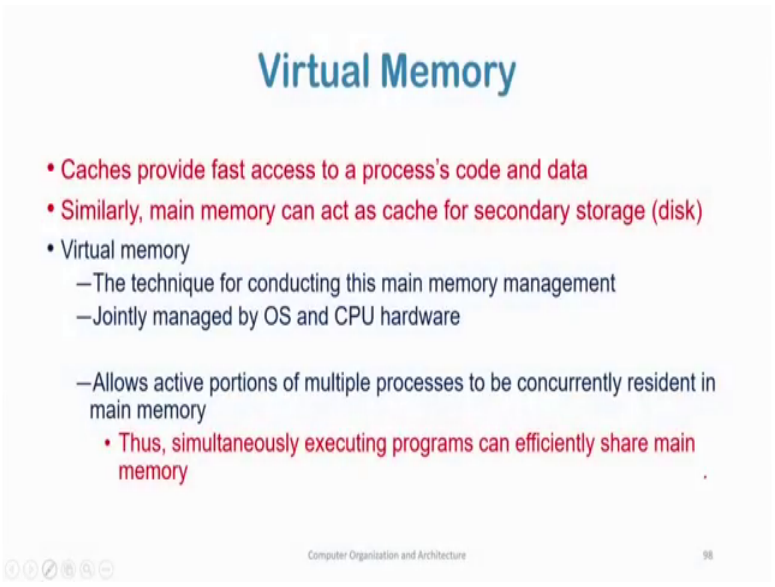
# Computer Organization and Architecture: A Pedagogical Aspect

**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

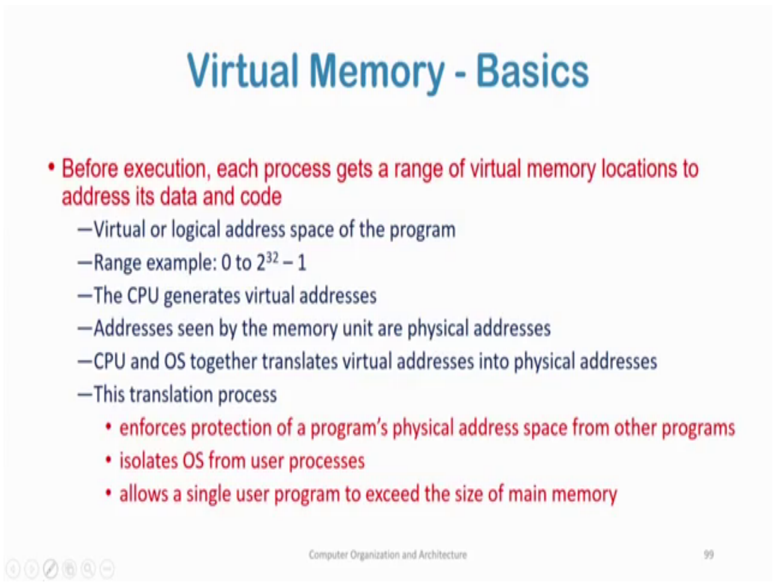
- Caches provide fast access to a process's code and data
- Similarly, main memory can act as cache for secondary storage (disk)
- Virtual memory
  - –The technique for conducting this main memory management
  - –Jointly managed by OS and CPU hardware
  - –Allows active portions of multiple processes 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 provides 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 to the power 32 minus 1. So, each process in the in the in the virtual memory managed system can access as big a memory as 2 to the power 32 minus 1. So, therefore, the each program or 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 to protection of a programs physical address base 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 to the power 32 minus 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 2 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.

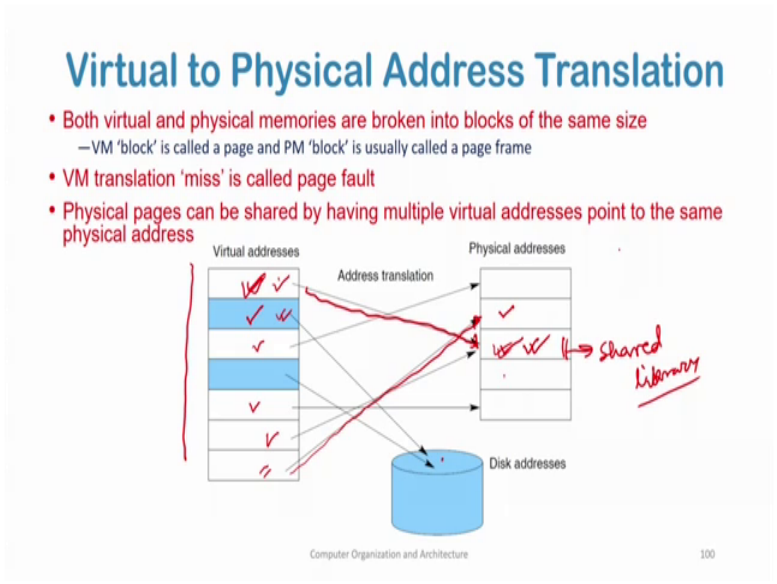
It also isolates the operating system which is which is a separate program from other user processes. Because the operating system is protected, it conducts and nicely executes all conducts all the processes happening in the entire system. So, that has to be protected from user programs OS runs, in kernel 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 to the power 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. We 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 lead 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)



## Virtual to Physical Address Translation

- Both virtual and physical memories are broken into blocks of the same size
  - —VM 'block' is called a page and PM 'block' is usually called a page frame
- VM translation 'miss' is called page fault
- Physical pages can be shared by having multiple virtual addresses point to the same physical address

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 divided 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, I during translation I have a 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 required the virtual page that I required 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 this 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 president 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 dynamic; dynamic link 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.

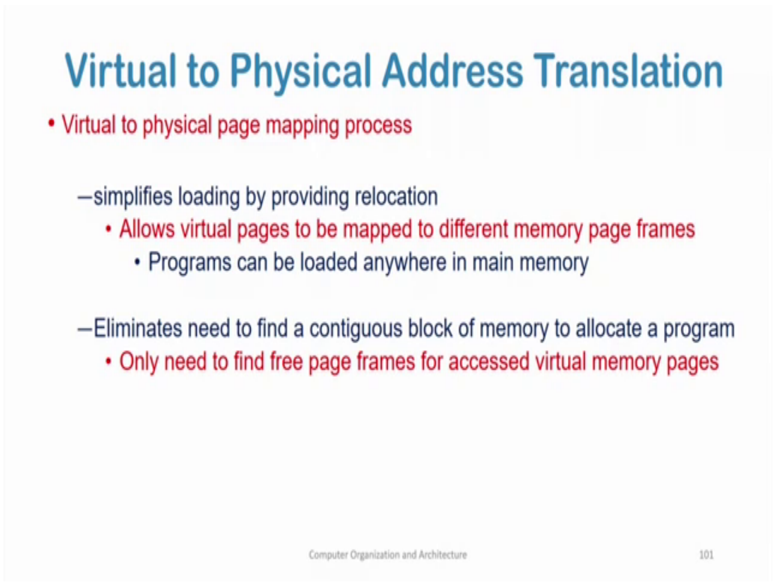
So, what happens is that for shared libraries you do not actually load the libraries as part of the as part of the address space in your in your executable code. Instead you have you just include a stub saying that printf is a shared library which I may need to access from a part of the program. Now, 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 if 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 we shared library code the 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

As we have seen in a virtually managed memory system the; a 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 a 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.

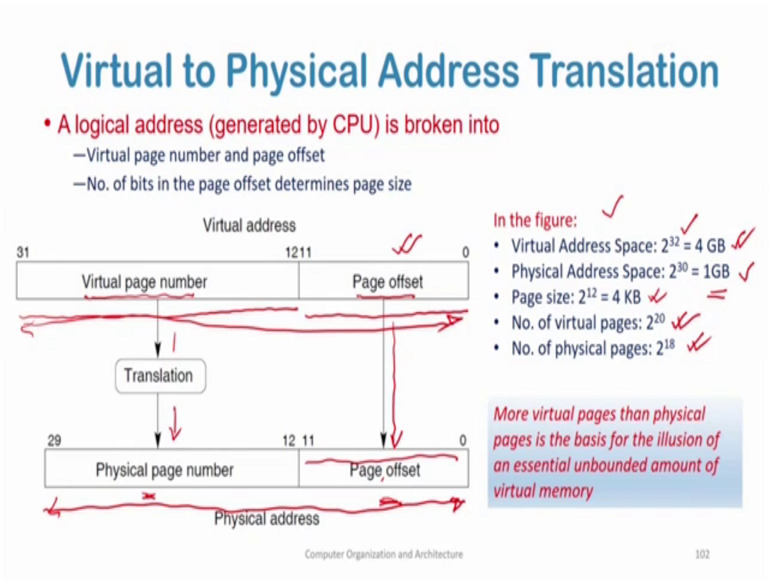
So, my in my virtual address space my code was contiguous one after another virtual page one followed by virtual page 2 followed by virtual page 3. However, virtual page 1 can reside in let us say page frame number 10, and virtual page 3 can reside in page frame number 1. There is no there is no issue 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)



## Virtual to Physical Address Translation

- A logical address (generated by CPU) is broken into
  - Virtual page number and page offset
  - No. of bits in the page offset determines page size

In the figure:

- Virtual Address Space: $2^{32} = 4$ GB
- Physical Address Space: $2^{30} = 1$GB
- Page size: $2^{12} = 4$ KB
- No. of virtual pages: $2^{20}$
- No. of physical pages: $2^{18}$

*More virtual pages than physical pages is the basis for the illusion of an essential unbounded amount of virtual memory*

Computer Organization and Architecture 102

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 to the power 12. 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 is let us say my CPU has asked for a data from a virtual address which is given by this 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 to the power 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 to the power 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 to the power 18. But the number of virtual pages that I can have for this process is 2 to the power 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 to the power 32. Because my I have 2 to the power 32 I have 2 to the power 32 different addressable locations in the virtual memory. The physical address space is of 2 to the power thirty from 0 to 29.

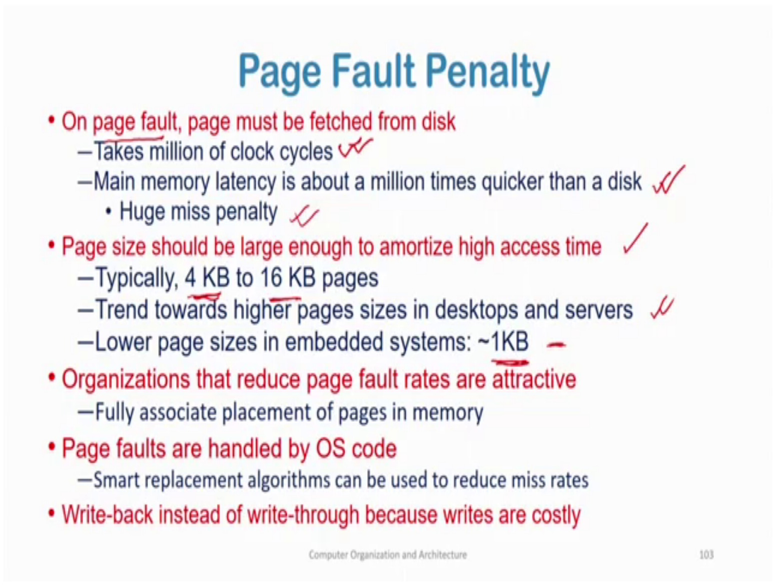
So, the size of the physical memory is 1 GB, the size of the virtual memory is 4 GB, the page size is 2 to the power 12 which is 4 KB, 2 to the power 10 is 1 KB.

So, 10 bits 1 KB and 2 bit small so 4 KB's the number of virtual pages as we discussed is 2 to the power 20 and the number of physical pages is 2 to the power 2 to the power a number of physical pages is 2 to the power 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 60, 64 bit address bus is in the 64 bit computer, then the number of the virtual address space can be as large as 2 to the power 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.

(Refer Slide Time: 24:56)



## Page Fault Penalty

- On page fault, page must be fetched from disk
  - Takes million of clock cycles
  - Main memory latency is about a million times quicker than a disk
    - Huge miss penalty
- Page size should be large enough to amortize high access time
  - Typically, 4 KB to 16 KB pages
  - Trend towards higher pages sizes in desktops and servers
  - Lower page sizes in embedded systems: ~1KB
- Organizations that reduce page fault rates are attractive
  - Fully associate placement of pages in memory
- Page faults are handled by OS code
  - Smart replacement algorithms can be used to reduce miss rates
- Write-back instead of write-through because writes are costly

Computer Organization and Architecture 103

Now page as I told that if for a corresponding virtual and corresponding virtual page number the physical page is not there is there is not a proper translation of the virtual page to a physical page, I have a page fault. What does that mean I have loaded a virtual address, for that I have a virtual page number, the translation told me that corresponding to that virtual page number this virtual page does not currently reside in physical memory.

So, I have to access the data corresponding data or code corresponding to this page I have to first bring this page from the secondary storage to a page frame in physical memory and then I will get a proper translation and after that can I access data in that physical page frame.

So, when I when I do not have the data in the physical page corresponding to a virtual address I have a page fault. Now page faults can be the page fault penalty for virtual memories is very high. Why because the access times on the secondary storage is very high. Now to access a, whereas let us say for accessing the main memory I will only take around say 50 to 70 nanoseconds for accessing the accessing the secondary storage I may take millions of nanoseconds.

For example it could be as big as 5 million nanoseconds let us say. So therefore, to access the secondary storage to a service a page fault I need to the secondary storage to fetch a page from the disk and this takes millions of clock cycles. The main memory latency is about a million times quicker the main memory latency is about a million times quicker than a disk.

So, this amounts to a huge miss penalty. So, therefore, a lot of effort has gone into trying to reduce misses. So, what are the kinds of efforts that has gone into now the first one is that the page size that we decide like we have to decide what should be the size of a block of cache.

Similarly we have to decide what should be the size of a page. Now page sizes should be large enough to amortize the high cost of accessing the secondary storage. So, once I access the secondary storage I need to need to bring a lot of data together to maximize locality of reference as much as possible. So, if I bring all a big sized page from the secondary storage if the probability is that for further accesses the data.

If the data is clustered around the address that I currently fetched then subsequent accesses will not result in page faults if the page size is big. Also the other part is that when for magnetic disks for example, secondary storage the access times for a for numerous for numerous accesses is much larger than if I have bigger contiguous access of the data bigger contiguous access of a significant amount of data.

If I have that is much better than having multiples multiple numerous accesses to smaller amounts of data due to the organization of the secondary storage. So, therefore, or both these aspects some lead to the conclusion that the page sizes should be large. Typically today page sizes are of the order of 4 KB's to 16 KB's. The trend newer trends for desktops and servers are that it is going to be still higher to even say 32 KB's or 64 KB's. So, each page will be off size say since 32 KB's or 64 KB's why to in order to reduce the number of times I need to go to the secondary storage to access the to access data.

So, once I go to the secondary storage I want to bring a lot of data together. So, that further accesses for further accesses of data I the probability that I have to go to the secondary storages minimize the probability of a page fault is minimized.

However, for embedded systems page sizes are typically lower of the order of 1 KB. This is this is one reason for this is that it could be that embedded systems are resource constrained. So, their memories are lower and bigger the page sizes bigger becomes the internal fragmentation of the last page. Suppose for example, I have a page sizes of 4

KB, and I 4 KB and I have say and I have a process whose virtual address space whose virtual address space is of size 18 KB.

So, I will require 5 pages 4 KB, 5 pages right, but the last, but in the last KB 2 k in the in the last page 2 KB of space will be wasted. Because I required only 18 KB, 2 KB of space will be wasted this amounts to internal fragmentation, and in many embedded systems may not afford such internal fragmentation.

And also because for embedded systems the processes that are executing and their memory accesses are much more predictable I can have more efficient ways of managing memory in embedded systems. Because the exact processes that will be run are typically known this is not so for desktop and servers arbitrary processes can come at any point in time and execute. So, in resource constrained embedded systems page sizes are typically lower. Organizations that reduced page fault rates are also attractive. So CPU system organizations that reduce page fault rates are also attractive in virtual memory managed systems.

So, virtual memories are typically use fully associative placement of pages in main memory. Now for caches we saw that the principle; there were 2 principal problems of having fully associative caches. One was that I needed more expensive hardware to search for the tags to search for a tag match which has to be done in parallel ok. Because a particular physical memory block can reside in any cache line. So, I have to check for the tags of all the cache lines to find out where my data exists. So, this made it on this made is it very expensive for cache memories.

The other way the other problem was that when I needed a replacement how to find which page to replace the to find let us say the LRU page, the least recently sorry which block to replace to find the least recently used block I will need a very expensive hardware to keep track of which page is least recently used at a given time.

So, we used typical use set associative mapping for caches. Now for physical for virtual memories on the other hand associative placement is more beneficial. Why is it more beneficial? Because the access times to the secondary storage is very large. Now so what I want to minimize page faults I need I want the virtual page to I want a virtual page to I want to be able to map a virtual page to any place within the physical memory to any page frame within the physical memory wherever possible. Because that minimizes my misses page faults. Now this is much lower.

Now what is the drawback of this approach the search of where a virtual page exists in the physical memory becomes higher, but this search is much lower in cost in comparison to the cost of a page fault.

Hence the fully associative placement of pages in memory are preferred for virtual memories and page faults are handled by the OS code, and not hardware. Now why is this so, because so this is being done in software. So, again this will be more costly as compared to hardware; however, because page faults are very expensive compared to that this the compared to that handling page faults in software is much lower in cost.

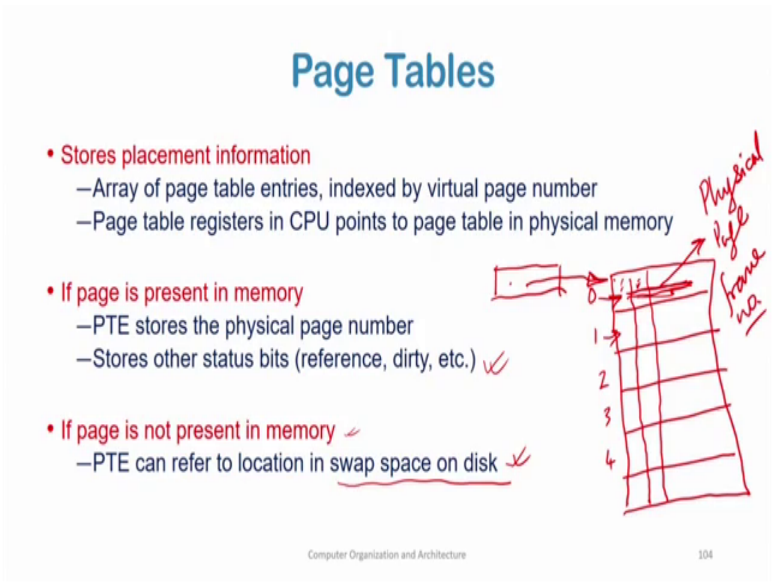
In addition to this when I handle page faults in software smart replacement algorithms can be used to reduce cache misses. We will look at different replacement algorithms and how such replacement algorithms helps reduce cache sorry not cache misses page faults. So, smart replacement algorithms can be used to reduce page faults and we will see how the replacement algorithms allow help reduce page faults.

And obviously, the last point here says that right back caches are sorry right back mechanism is used for in virtual memories and not right through which means that suppose when I write on 2 a physical memory I do not write to the to the corresponding location in virtual memory or the few or the secondary storage. I use a write back scheme because if I go on each time I write into physical memory if I have to write into secondary storage it will be hugely costly as we understand. So, right through is cannot be used right back is used.

So, whenever the page needs to be replaced modified pages will be replaced and if the page has is dirty or modified is written to when it is being replaced it will be placed into the secondary storage. And we will not use write through because as we understand write through will be very costly. And write through for each write into the physical memory I must also write into the secondary storage which is not possible affordable at all.

Now, we looked in more details as to how virtual address to physical address translation is done. This translation is done through a data structure called page tables.

(Refer Slide Time: 37:46)



## Page Tables

- Stores placement information
  - Array of page table entries, indexed by virtual page number
  - Page table registers in CPU points to page table in physical memory
- If page is present in memory
  - PTE stores the physical page number
  - Stores other status bits (reference, dirty, etc.)
- If page is not present in memory
  - PTE can refer to location in swap space on disk

Physical Page frame no
0
1
2
3
4

Computer Organization and Architecture 104

So, page table stores placement information it has an array of page table in entries indexed by virtual page number. So, for each process I will have a page table this page table will have these are virtual page which have we will have entries for each virtual page. So, virtual page number is 0 1 2 3 4. So, these are virtual pages.

Corresponding to each virtual page the page table will contain the corresponding physical page frame number physical page frame number ok. So, page tables store the placement information it is an array of page table entries it is an array of page table entries indexed by virtual page number.

So, the index is the virtual page number and it contains what the physical page number. So, page table registers in CPU stores in CPU points to the page table in physical memory. So, when I have a contact switch what do I do during a context which a new process is executed on the processor. With during that contact switch for each process I will also have a page table which maps what it us virtual pages to its physical pages this physical this page table is also resident in main memory.

Now the page table register is a hardware register which during a contact switch is populated with the address of the starting address in which is populated with the starting address of the page table in physical memory. So, again page table is an array of page table entries it is a data structured which is there per process every process has its own page table. Page table maps the virtual pages corresponding to that process to corresponding physical pages ok.

Page table in page table is an array which is indexed by virtual page number and the page table register is a hardware register which contains the address of the start of the page table. So, the address of the start of the page table in physical memory is kept in the page table register. So, the data corresponding to this process and also the data code of this process is in physical memory and the page table of this process which maps virtual page numbers to physical page number this is also kept in the physical memory.

So, and there is a distinct or its own each process has its own page table. So, during a contact switch what do I do? I populate this hardware page table register with the address with the starting address of my page table of this processes page table which has come to the processor subsequent to the context switch the starting address of this page table is put into the page table register. If a page is present in memory PTE, stores the physical page number, and it can stores other bits such as reference bits, dirty bits etcetera.

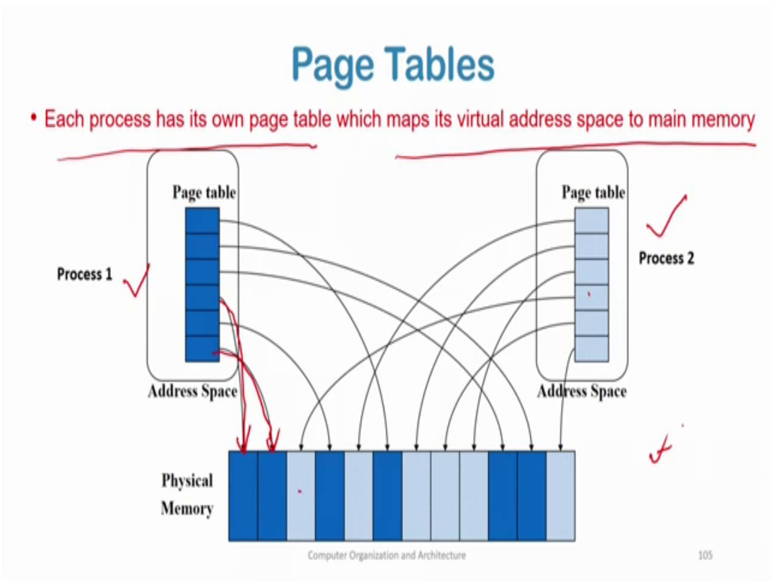
So, we will talk about all this later reference bits are used to understand whether this page was referenced within a given time in the past. So, I have different time epochs and within a given time interval if within a particular specified time interval. If my page was accessed particular time interval from the current time to the past within a given time in the past interval if my page was accessed right referenced bit will 1. Otherwise may referenced it will be 0. Why that is used will come later the dirty or modified bit is used to understand the dirty or modified bit is used to understand whether this page was written to or not.

Why is this required as I said this virtual memories are right back. So, if I find that the dirty bit is on and I want to replace this page, then what do I need to do I have to if this page is dirty then I have to write it back into the secondary storage. So, we will look at these later in more details if a page is not present in memory this will be a page fault the page table entry can refer to location in swap space.

So, I have to go to the swap space in disk on the disk and bring back this page. So, the swap space is something called as we know in Linux you must have heard the term swap partition this swap partition is basically the swap space with that we are talking about. Why this is there in the swap space typically contains the portions the data portions of each process which are dynamically changing. The code part which will never change the code of the process is never going the code of the processes is never going to change that is typically kept in the other part of the secondary memory along with the process.

The data which will be frequently changed is put in a in a special partition in a special small marked area within the secondary storage which will be which we called the swap space on the disk.

(Refer Slide Time: 44:07)



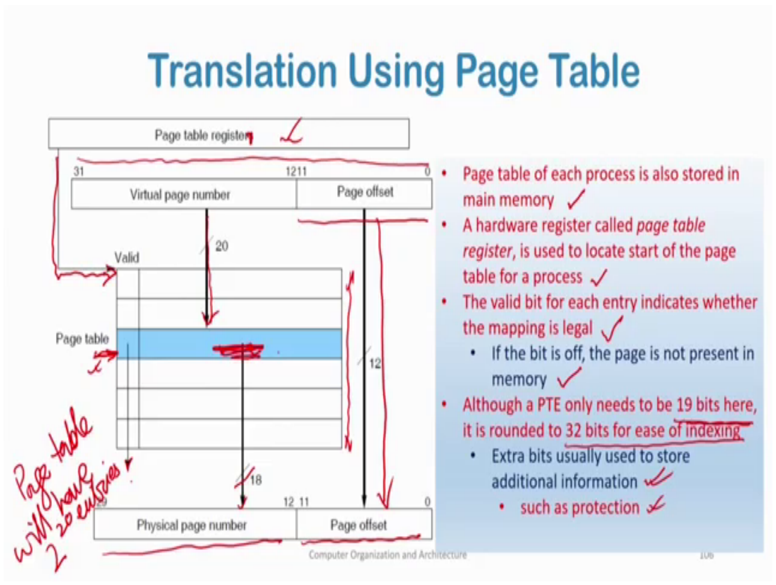
## Page Tables

- Each process has its own page table which maps its virtual address space to main memory

So, as I said each process has its own page table, and this maps its virtual address into the physical memory. So, here if this is the page table of process 1, this is the page table of process 2. And we see that the pages the pages are mapped to corresponding page frames in the physical memory. And the physical memory is nicely shared between these process 1 and process 2.

A few of the page frames contain data corresponding to process 1, other page frames contain data corresponding to process 2 ok. This is this figure shows how nicely this physical memory is being shared by 2 processes through their own page tables.

(Refer Slide Time: 44:58)



## Translation Using Page Table

- Page table of each process is also stored in main memory ✓
- A hardware register called *page table register*, is used to locate start of the page table for a process ✓
- The valid bit for each entry indicates whether the mapping is legal ✓
  - If the bit is off, the page is not present in memory ✓
- Although a PTE only needs to be 19 bits here, it is rounded to 32 bits for ease of indexing
  - Extra bits usually used to store additional information ✓
    - such as protection ✓

Computer Organization and Architecture

Now we will see how this translation occurs. So, now this is with respect to a single process. So, I have a single page table register there is a single page table register this page table register contains what it contains the start or starting address of my page table in physical memory ok.

So, now what happens the CPU when this contact switch was done? This page table register was populated with the address of the starting address of my page table. So, now I can access my page table I can access this page table. Now the CPU the program is running the CPU has generated an address for a let us say for a given data and that address is a virtual address. So, this is my virtual address.

Now this virtual address will be mapped the offset part will be directly will be directly used, in the physical address as well the offset part of the virtual address will be directly used in the physical address as well without no modification. Now I have this 20 bit virtual page number let us say now for each entry in this for each entry 2 to the power 20. There are 2 to the power 20 different virtual pages for each virtual page I will have an entry in the page table.

So, what will be the length of my page table my page table will have 2 to the power 20 entries page table will have 2 to the power 20 entries ok, we will have 2 to the power 20 entries. Now let us say corresponding to this virtual page number these 20 bits, this is the index, and this is the physical page number from the valid bit I have found that what I have found that this mapping is valid which means that corresponding to this virtual page number this physical page number is actually valid.

The data corresponding to this virtual number actually resides in this physical page number. If this valid bit was off it means that this page number corresponding to this virtual page number this physical page is not there in physical memory. So, I will have a page fault if this is off I will have a page fault. Now if this is on let us say this is on then what happens for this virtual page number I will get the physical address physical page number this physical page number is of 18 bits.

And I get the higher order 18 bits of my physical address and I also get the page offset I add this up together and get my 30 bit physical address. So, the page table of each process is also stored in main memory. A hardware register called page table register is used to locate start of the page table for a process. The valid bit for each entry indicates whether the mapping is legal if the bit is off the page is not present in memory main memory although a page table entry needs only 19 bits here.

Why do we need only 19 bits here? I have 18 bits are required for my physical page number and 1 valid bit. So, in this case I required only 19 bits; however, I typically round it to 32 bits for ease of indexing. So, I typically round it to the next power of 2 for ease of indexing because I work with bits I can break addresses and I can do many things. However, although I waste a number of bits in my page table entries I still round it up to 32 bits. These extra bits; however, are in many times used for many for to keep many a types of additional information for example, protection information.

So for example, for a given particular virtual page I know whether this virtual page contains a programs code, or the programs data. If it is a programs code I can keep an additional bit to say that this part is read only I should not be able to I should not be able to modify this part of my this part of my data. So, if this is a code then I should not modify the corresponding content in my main memory if corresponding to this virtual page so that information I can keep.

So, even if the CPU tries to modify the data I will say that this is an invalid operation based on protection bits. Similarly I can keep read, write, execute bits different other protection bits, corresponding to using these free bits that I have that are unused. So, additional information can be used for in these bits on a per page basis for each page I can keep separate protection bits. So, I can use it in this way.

With this we come to the end of this lecture.