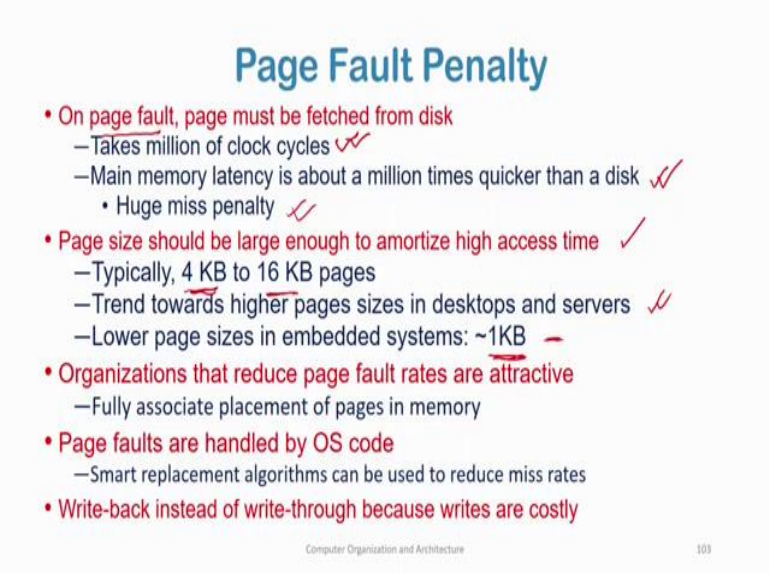
(Refer Slide Time: 24:56)



Now page as I told that if for a corresponding virtual 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 don't 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 ok.

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 a big sized page from the secondary storage 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 for magnetic disks for example, secondary storage the access times 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 multiple numerous accesses to smaller amounts of data due to the organization of the secondary storage. So, therefore both these aspects 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 of size say 32 KB's or 64 KB's why? In order to reduce the number of times I need to go to the secondary storage to access 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 storage is minimized 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 so.

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 reduce 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 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.

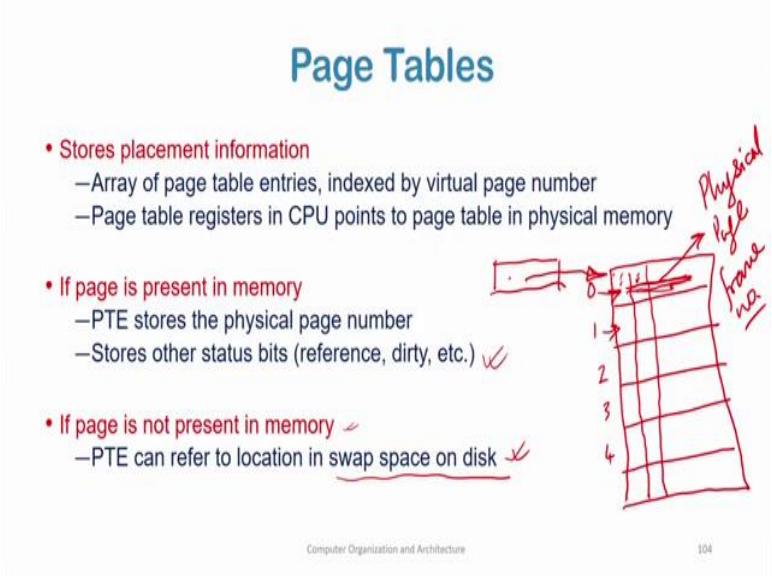
In addition to this when I handle page faults in software smart replacement algorithms can be used to reduce cache misses. We will look at different replacement algorithms and how such replacement algorithms helps reduce cache sorry not cache misses page faults. So, smart replacement algorithms can be 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 write back caches are sorry write back mechanism is used for in virtual memories and not write through which means that suppose when I write on to a physical memory I don't write to the to the corresponding location in virtual memory 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, write through is cannot be used write back is used.

So, whenever the page needs to be replaced modified pages will be replaced and if the page 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 look 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)



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 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 context switch what do I do? During a context which a new process is executed on the processor. With during that context switch for each process I will also have a page table which maps what its 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 context 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 structure 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 context 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 ok. 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.

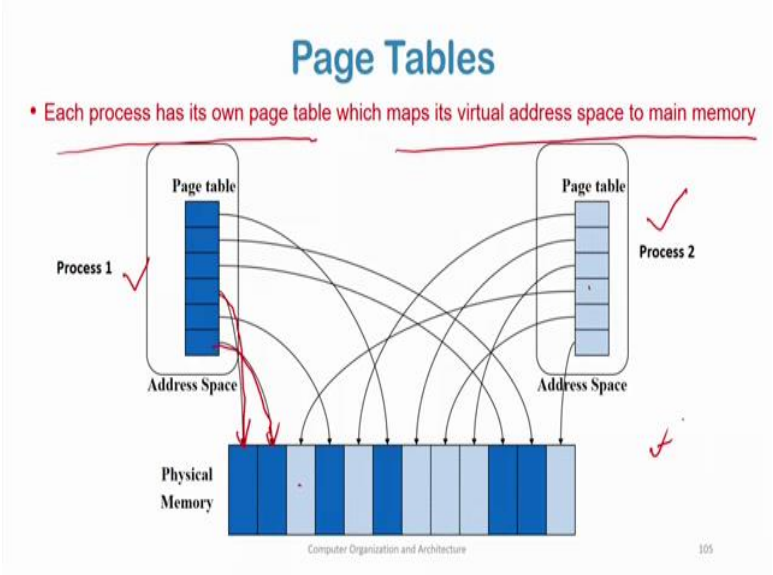
So, we will talk about all this later reference bits are used to understand whether this page was referenced within a given time in the past. So, I have different time epochs and within a given time interval if within a particular specified time interval. If my page was accessed particular time interval from the current time to the past within a given time in the past interval if my page was accessed my reference bit will 1. Otherwise my reference it will be 0. Why that is used we 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 write 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 ok. 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 called the swap space on the disk.

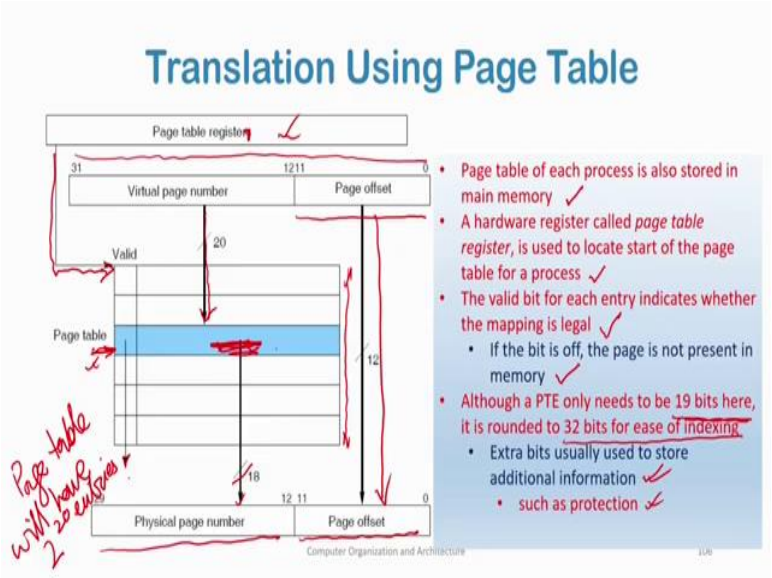
(Refer Slide Time: 44:07)



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)



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 context 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^{20}. There are 2^{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^{20} entries page table will have 2^{20} entries ok, we will have 2^{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 ok and I also get the page offset I add this up together and get my 30 bit physical address ok. 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 require 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 to keep many 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 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 ok. 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.