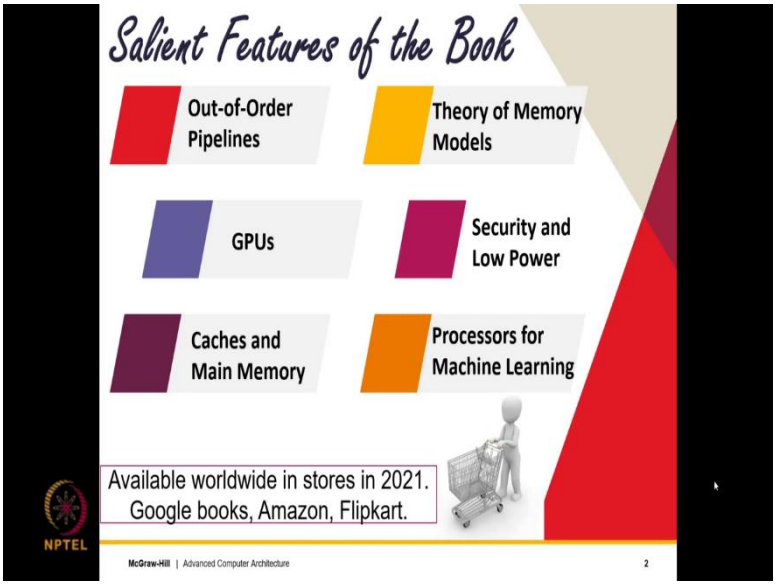
Advanced Computer Architecture

Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

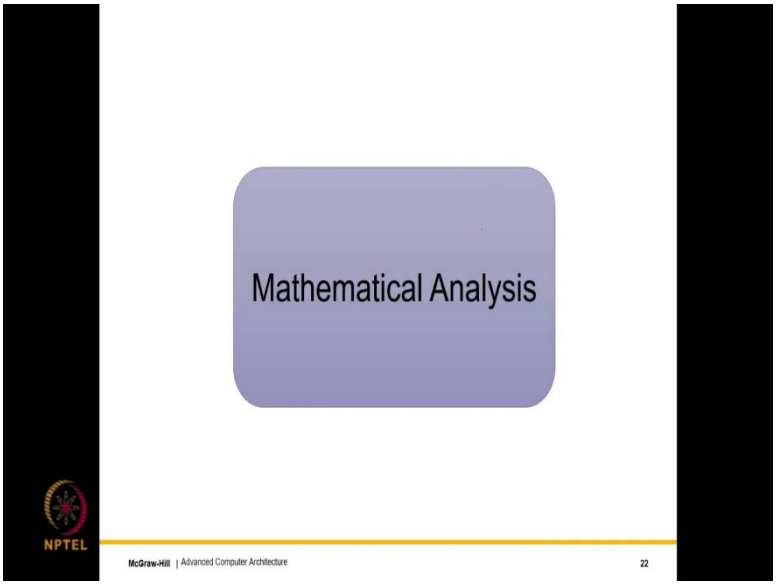
Lecture - 20
Caches Part - II

(Refer Slide Time: 00:17)

(Refer Slide Time: 00:36)

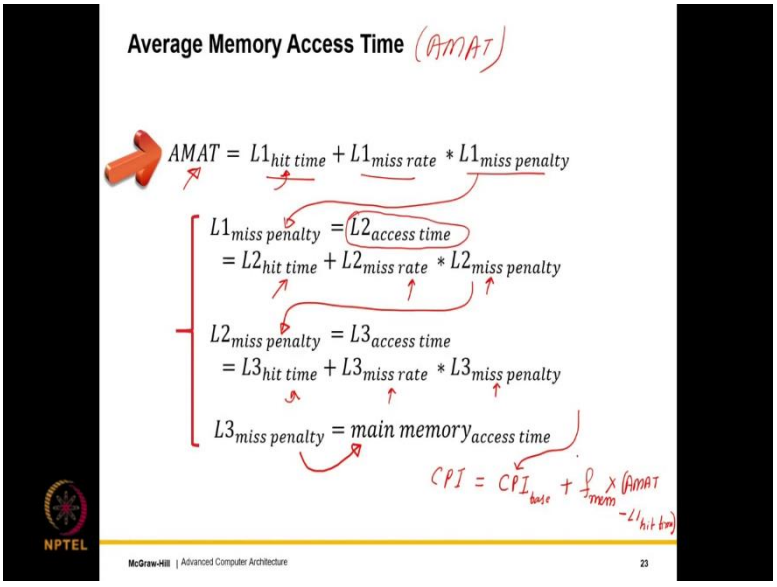


(Refer Slide Time: 00:36)



Now, that we have seen the design of the cache, let us do a little bit of mathematical analysis of the functioning of the memory system.

(Refer Slide Time: 00:44)



So, let us define the average memory access time, the AMAT. So, the average memory access time is basically defined like this, it is the L1 hit time. So, regardless of a hit or miss you will always do a lookup. So, only then you will get to know whether it is a hit or miss. So, the hit time is something you have to pay plus the L1 miss rate times the L1 miss penalty. So, this is simple math comes from the formula of expectation.

Now, the L1 miss penalty is roughly the same as the L2 access time of course, there might be a little bit of a delay in terms of communication from L1 to the L2, let us ignore that for the time being. The L2 access time is essentially the L2 hit time for the same reason, let us say L2 miss rate times the L2 miss penalty.

The L2 miss penalty if there is an L3 is again L3 hit time + L3 miss rate * L3 miss penalty. And finally, the L3 miss penalty it will go to the main memory. So, the main memory you assume that you will always have a hit. So, the so, there is no miss time or miss penalty with the main memory. So, this will be the main memory access time. So, this is the formula for the average memory access time or the AMAT.

And the way that the AMAT is related to the CPI is that the CPI log cycles per instruction is basically the CPI with the ideal memory. So, let us call it CPI base times the fraction of memory instructions times AMAT - 1 or let us call it AMATs. I mean I would not call it the AMAT - 1, but AMAT - the L1 hit time.

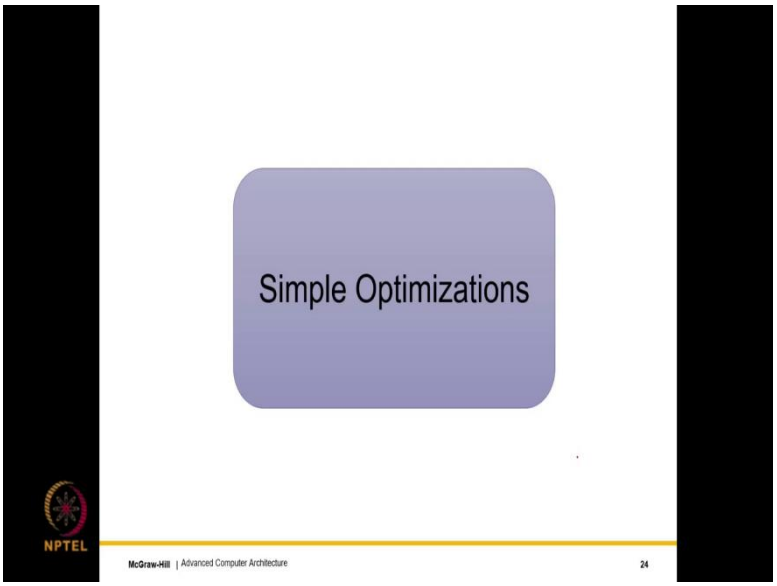
So, let us analyze this for a second. So, let us assume that the L1 miss rate is 0. See you never miss an L1. See you never miss in the L1 the AMAT = the L1 hit time and a CPI = the CPI base. So, which is what I was defining as an ideal CPI which means that you always hit in the L1 cache. Let us define this as a CPI base.

Now, consider the practical scenario where we have L1 misses. So, in this scenario with L1 misses the AMAT will not be equal to the L1 hit time, it will be more than that. So, that is why we need to change the CPI. So, by the simple formula of expectation this will get multiplied with a fraction of memory instructions. So, fraction of memory instructions, which are on the critical path.

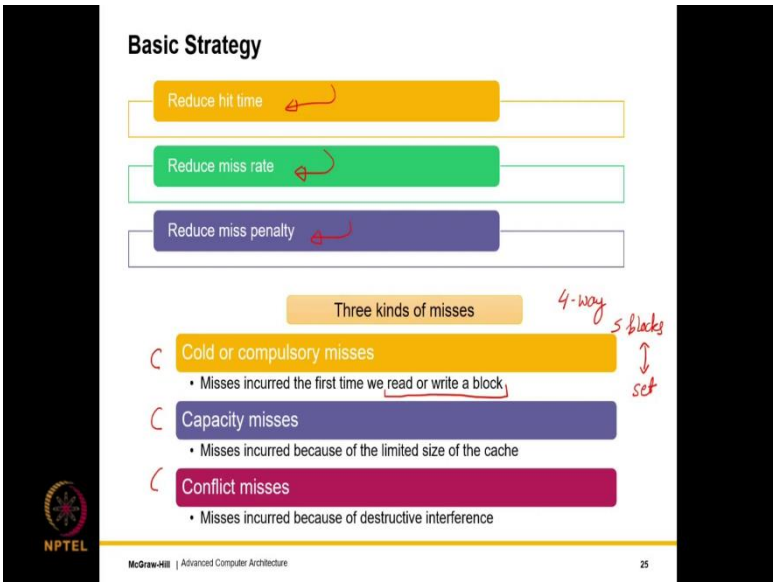
So, of course, this formula is more relevant for in order pipelines and out of order pipelines because there is a fair amount of overlap in terms of the miss times of different memory instructions. So, that is also one advantage of an out of order pipeline nevertheless we will multiply the fraction of memory instructions f mem into AMAT - the L1 hit time.

So, for us this is the math of computing the CPI. Again this is primarily in order processor formula for the hit time miss rates and miss penalties.

(Refer Slide Time: 04:38)



(Refer Slide Time: 04:42)



A few simple optimizations; so, simple ones. So, what is the basic strategy of having a having an efficient cache? We either reduce the hit time; we either reduce the miss penalty sorry the miss rate or we reduce the miss penalty. Any one of them or any combination there off. There are three kinds of misses we can have. So, regardless of the optimization strategy there are three kinds of misses.

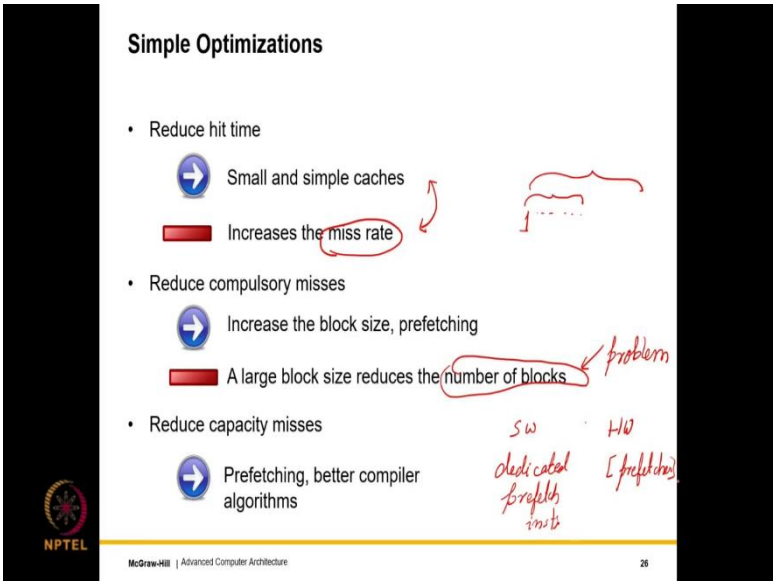
So, they are called the three C's because all of the misses start with a C. So, the first category of misses are cold or compulsory misses. So, these misses happen the first time

we read in data. So, the first time that we read a block and it is not there in a cache this miss will happen this is called a cold miss or a compulsory miss because this had to happen it was not there.

Capacity miss happens when because of the limited size of the cache. So, let us say that we frequently access 100 kilobytes of data, but our cache is only 64 kilobytes then for the remaining 36 kilobytes we will have a capacity miss. So, the way that way to solve capacity misses is to actually have a bigger cache, but this has its shortcomings. Finally, we will have conflict misses. So, conflict misses happen because of the limited associativity of the cache which are caused because of destructive interference.

So, consider let us say 4 way set associative cache. There might be 5 blocks that map to the same set. If 5 blocks mapped to the same set, then there is a problem because all 5 cannot be stored. So, continuously they will be displacing one and that will lead to a performance loss. So, such kind of misses are known as conflict misses. So, as you can see these are three C's cold capacity conflict or compulsory capacity and conflict. So, these three types of misses are what we have to optimize for.

(Refer Slide Time: 06:56)



So, some simple optimizations for reducing the hit time we can have small and simple caches. So, basically the have a small and simple caches the hit time will get reduced, but the negative aspect is it will increase the miss rate. Smaller is the cache higher is the miss rate. So, we probably do not want this.

So, some trade off has to be struck between these two factors. To reduce compulsory misses one thing that we can do is we can increase the block size. If we increase the block size, so, then instead of fetching let us say 64 bytes in one go if we fetch 128 bytes in one go. This will ensure that because a spatial locality. We can reduce the number of misses that would have happened because of a compulsory nature. And recall that spatial locality means that we will access nearby addresses with a very high likelihood.

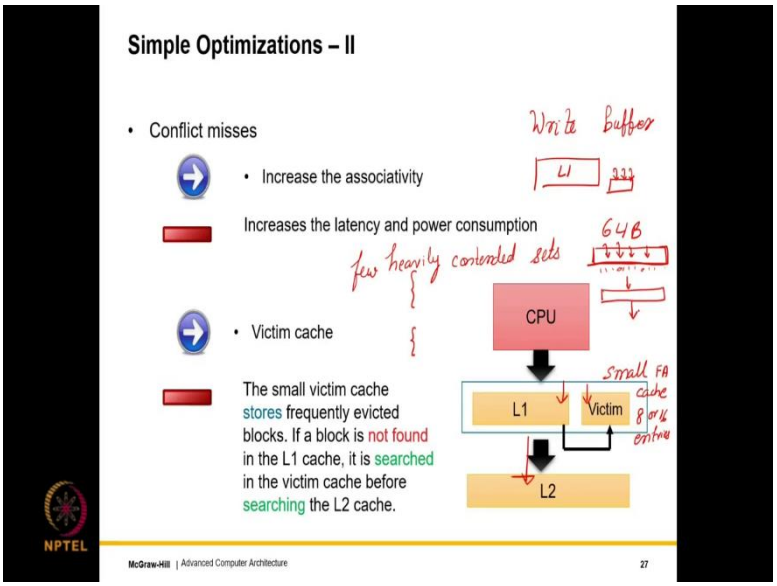
So, let us say we access some byte. The nearby bytes we will access. See if our block size is small we will fetch this much if our block size is large we will fetch this much. So, larger block size is clearly beneficial, but again a large block size reduces the number of blocks that we can store in a cache which is also a problem.

So, basically if we store few blocks then we are not covering a large part of the address space which is an issue. To reduce capacity misses when you can have a bigger cache, but a bigger cache is slower. We can do what is called prefetching which means that we can predict the addresses that we will fetch in the future or the addresses that we will send to the memory system in the future and then we can solve this problem either in software or in hardware.

So, in software we can insert dedicated prefetch instructions. So, dedicated prefetch instructions can be inserted in software. So, they are going to prefetch data for us and otherwise we can have a hardware unit called a prefetcher which we will study in great detail towards the later part of this chapter.

So, the prefetcher is an is a unit similar to a branch predictor. It predicts the addresses that we will request from the memory system in the future and it tries to fetch them in advance such that they are already there in the cache before we actually access the memory addresses.

(Refer Slide Time: 09:59)



Conflict misses, well simple idea is to increase the associativity, but again increasing the associativity has issues with latency and power consumption. So, here also prefetching helps, it is definitely beneficial. Another approach to reduce conflict misses is to use a victim cache. So, the key insight is so, in architecture we always need a key program level insights.

The key program level insight is that there are only a few sets that are heavily contended. So, only a few sets will have a lot of contention and so, then there will be some of these unfortunate blocks that will keep on getting evicted repeatedly. So, instead of storing them in the L2 cache why not have a small fully associative cache? So, in one of the rare examples of fully associative caches, a small fully associative cache which may be let us say has 8 or 16 entries.

So, then we here we store some of those blocks that account for a disproportionate number of misses which are like one of those unfortunate blocks in heavily contended sets which get repeatedly evicted. So, they can be stored in the victim cache. So, whenever an, whenever a block is not found in L1 cache it is searched for in both the L1 cache as well as the victim cache associated with the L1. So, we search in both. Only when we do not find the block in both do we actually access the L2 cache.

So, there are many many algorithms to manage the victim cache or manage which lines actually go to the victim cache. So, there are many such algorithms. So, we will not discuss

all of them here and we will you know leave many of that for the later sections of this chapter.

Another simple optimization which I am not mentioning, but might be worth talking about is the write buffer. So, in the L1 cache what happens is that let us say there is a miss. So, what will happen is that whenever we are doing and then. So, let us say there is a miss for a line and then a lot of write requests start coming.

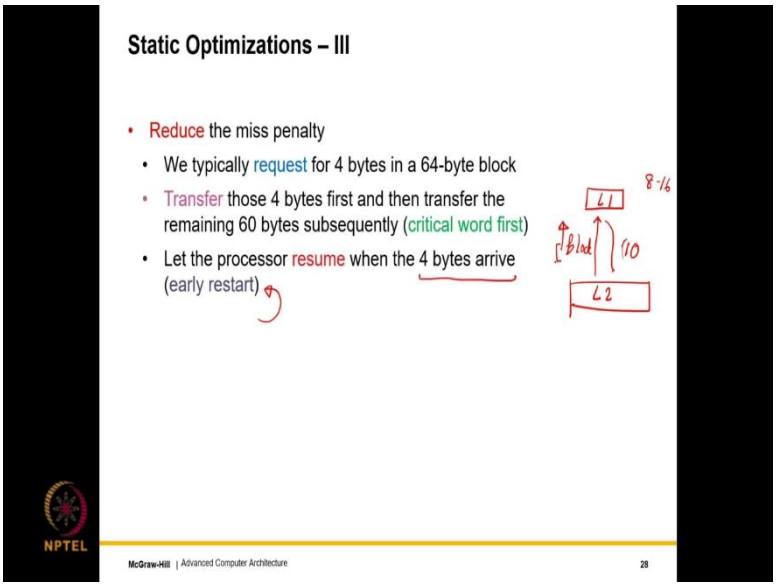
So, a lot of write requests start coming what we can do is that we can have a small buffer over here fair let us say there is a writes at two different addresses within the same block. We can just see keep collating those writes over here. So, consider 64 byte line and let us say that we write to the first 4 bytes middle 4 bytes later 4 bytes. So, why have separate write requests and why separately access the cache?

Instead what we can do is that we can maybe transfer the line to a small write buffer or without transferring the line what we can do is we can have a small separate structure where we simply take all of these writes. And we expect that these writes will come together in a small window of time because of spatial locality.

So, we can simply record the fact that look these are the bits that are being written and then we can collate all of those writes and then make one consolidated write. In a sense merge this with the contents of the line to get the final contents of the line So, the write buffer is one way of actually kind of absorbing different writes issued to the same block in the same window of time.

The advantage of this is that we reduce the number of cache requests and this helps us save power primarily. This primarily and also helps us reduce the traffic to caches. So, both reduction of traffic as well as saving of power by kind of locally collating the writes within a small window of time. So, this structure is known as a write buffer which can again be implemented as a small fully associative cache.

(Refer Slide Time: 14:51)



How do we reduce the miss penalty? Well, another way of reducing the miss penalty along with the write buffer is that we typically request for. So, the insight is that we typically request for 4 bytes in a 64 byte block. So, the question is that how are they actually transferred.

If this is the L1 cache and let us say if this is the L2 cache and let us say it takes 10 cycles for the data to come and let us say the data comes in chunks of 4 bytes from the L2 to the L1.

Then the transfer itself will take 16 cycles. So, it will take 10 cycles for the first chunk to come, but subsequent chunks will gradually come later. But we typically have instead of 4 bytes we can even transfer 8 or 16 bytes per cycle, but then also the entire transfer will take somewhere between 4 to 8 cycles.

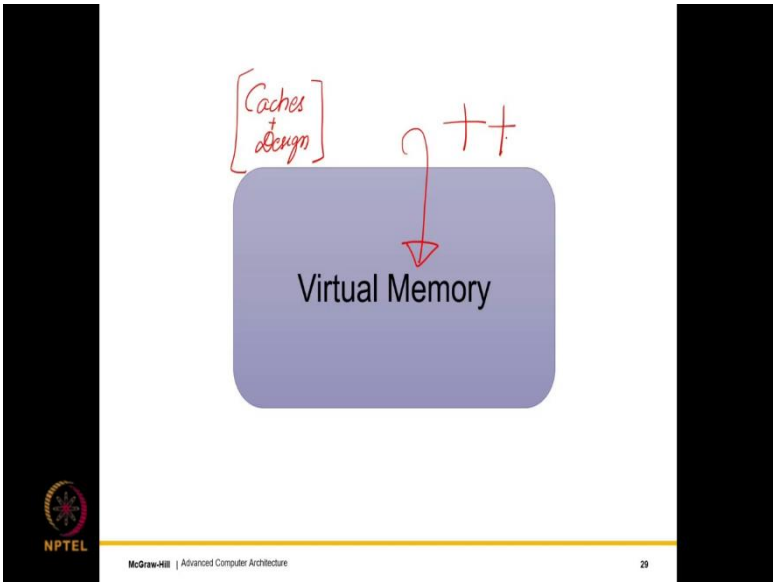
So, the question is that how do we order the bytes within the block while we are transferring from the L2 to the L1. If we transfer those 4 bytes first that are required by the processor which missed in the first place and then transfer the remaining 60 bytes subsequently then what will happen is that we can get the useful information earlier.

So, what I am saying over here is that do not transfer the contents or the block in order like from byte 0 till byte 63, instead send the data that was required by the processor first and the remaining bytes after that. Once the 4 bytes arrives the processor can do what is called

an early restart which means starts start the job of processing even before the entire block has arrived. So, this optimization is known as the critical word first early restart optimization.

And this helps us in improving performance by slightly reordering the way in which blocks are transferred. So, we transfer the slightly reordering the bytes in a block right the way it is transferred. So, we transfer those 4 bytes that are required first and we transfer the remaining bytes later in later cycles.

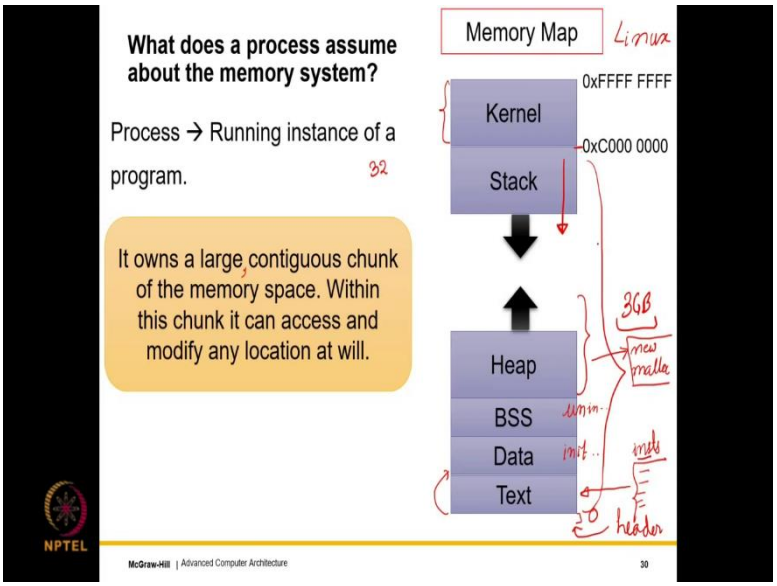
(Refer Slide Time: 17:19)



So, our discussion of caches has ended over here. So, we have kind of a basic overall bare bones understanding of the way that caches actually work. So, there are two parts to the undergraduate study of the memory system in caches. So, one is of course, caches and their design which is what we have looked at. We have not looked at from a thorough electrical point of view, but we have looked at it from an architecture point of view.

The other important concept which I would say is important with a double plus is virtual memory. The reason is that most people actually get it wrong even though it is very simple, but nevertheless 99% of the people do not seem to understand it. So, let us try to unpeel the layers of virtual memory over here.

(Refer Slide Time: 18:12)



So, before understanding what is virtual memory we should understand what is a process and what does the process assume about the memory system. So, a process is nothing but the running instance of a program. In the sense it is a program in execution. We furthermore, assume that the process owns a large, contiguous chunk of the memory space.

So, if not the entire memory space, but at least a very very large fraction of the memory space it owns it exclusively owns. Within this chunk it can access and modify any location at will without permission. So, let us look at a simple memory map of Linux. So, consider a 32 bit memory system.

So, of course, we do not have many 32 bit memory systems as of today. Most memory systems are 64 bits and, but nevertheless let us go back to the era of 32 bits. It will be kind of simpler to explain. So, 32 bits is basically can be represented as 8 x digits. So, the addresses would go from 0 to 0xFFFF 8 F's. So, this would be the largest address and this would be the smallest address.

So, what Linux assumes is that it gives the upper 25% upper 1 gigabyte of the address space to the OS kernel. So, we are not in a position now to say why, but the entire address range which is from 0xC0000 all the way down to 0 which is a good 3 gigabytes is entirely with a process. And a process is what? It is a running instance of the program.

So, we assume that it is the undisputed irrefutable owner of this entire address space and it can access any location and do anything it wants with this address space. So, in this address space since we have a downward growing stack, the stack starts at the highest address which is the highest address for the process of course, and the stack starts to grow downwards and as you can see the stack itself can be huge. So, the sum of the stacks of all the functions can be huge.

So, the well I mean it is not that different functions have a different stack. They all share the same stack region, but let us say that even I have very deep recursion my stack can grow and there is a lot of space for it to grow. So, this diagram is clearly not up to scale. And so, stack we can keep growing downwards, downwards, downwards for gigabytes.

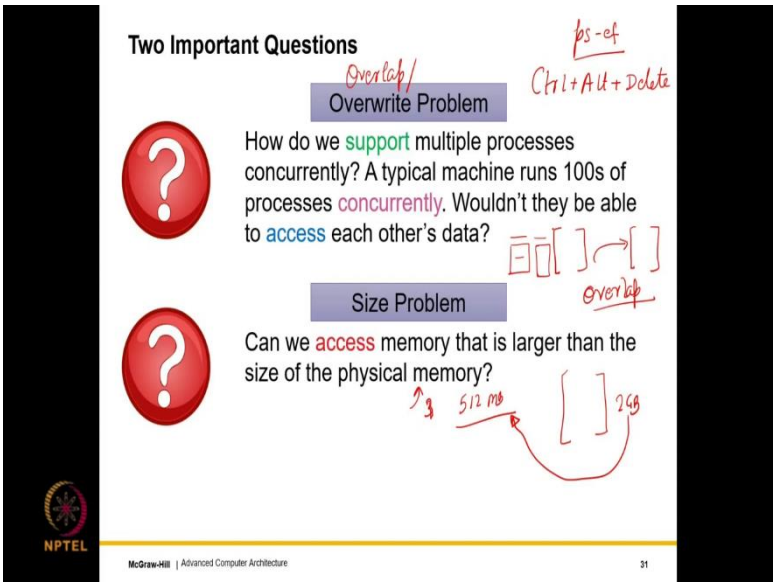
Now, let me start from the bottom. So, actually the first few bytes store what is called the header of the process. So, it has some data about the process contains something called a magic number and other information about the layout of the memory map. So, I will not discuss that that is for a separate course.

So, but you can see that the first few bytes maybe the first 100, 200 bytes are for storing the header information and then storing pointers to the different sections of course, but the important region for us of interest is a text region. So, the text region contains the instructions of the program.

So, the text contains all the instructions of the program and then we have two regions data and BSS. So, the data section contains initialized global and static variables and the BSS section contains uninitialized global and static variables. So, let us call it uninitialized and this is initialized. The heap section of the memory map contains all the data structures that are allocated via the new or malloc calls in C++ or Java where we dynamically create objects.

So, this is for dynamic creation of objects and these objects remain alive across function call invocations as opposed to objects that are allocated on the stack. So, these heap based objects are allocated once. So, they are located and freed using malloc and free are new and delete as is the case in languages such as C++ and Java. And so, the memory map is holds for every process. Every process assumes that it can write to the location 0 to 0xC000 0000 at will and all of these sections are accessible to it.

(Refer Slide Time: 23:42)



So, there are two problems. Now, that you have seen the memory map you can easily appreciate what I am going to say. So, there are two problems. So, just do Control Alt Delete on a windows system. You will see just go to task manager. You will see that there are 100s of processes running on your machine.

Type ps on Linux; ps - ef for more details or just ps or do ps - ef it will show you everything that is running on your system.

You will also see a lot of processes that are running in parallel on your system. So, how is this happening? Well, we have discussed this in the past the way that this is happening is that one process executes for some time then a timer interrupt comes then another process executes then a timer interrupt comes so on and so forth.

But then the question is that there will be a problem of overlaps. So, it is possible that different processes multiple processes can write to each other's data region.

So, they can read and write to each other's regions of data because after all the processes see the same memory map. So, what stops one process from writing two data structures that belong to another process? So, there is an overlap.

So, we are calling an overlap problem or an overwrite problem does not matter it is the same thing. And what we are saying is that look the memory system has to be separate for each process.

Otherwise what will happen is that one process can either maliciously or unknowingly write in the region of another process and since all the processes have the same memory map as you can see the all of that stack start from the same point.

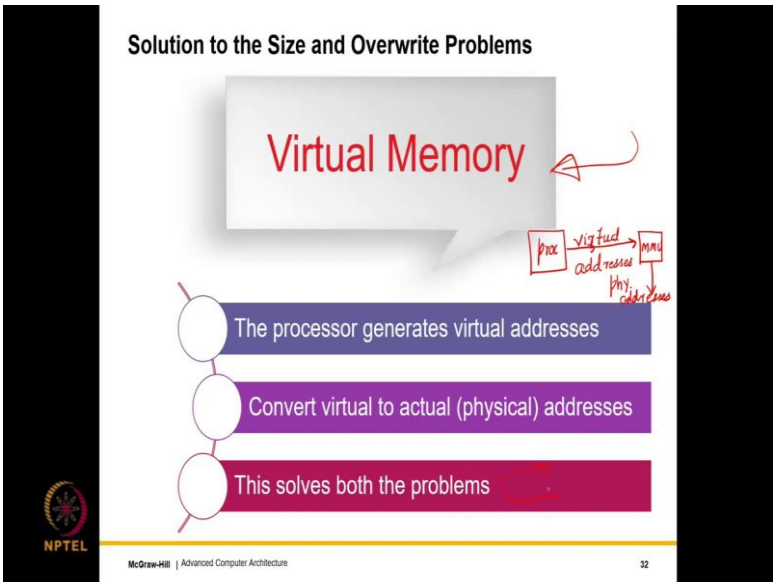
So, then it is very easy, for one write to actually corrupt the state of another process. So, this has to be stopped.

So, unless the processes actually write to different regions and memory we will not be able to stop this problem. The other is the size problem which means that can we access memory physical memory in this case which is larger than the size of the actual physical memory which is which we have limited to 3 gigabytes, but in reality let us say I only have 512 MB of RAM.

And my entire memory map maximum is 3 gigabytes but let us say, I am using 2 gigabytes of it. How will such a process run with so little main memory, which is half GB of main memory? So, can we not run such processes that would be really sad in the sense? That we are limited by our resources and even if we exceed them by a little bit, we will still not be able to run them. So, this is the size problem.

So, let me call this the overlap slash overwrite problem we will refer to it interchangeably and the size problem. Both of these need to be fixed to create a computer system out of just a processor. A computer system as you would recall runs multiple processes concurrently at the same time.

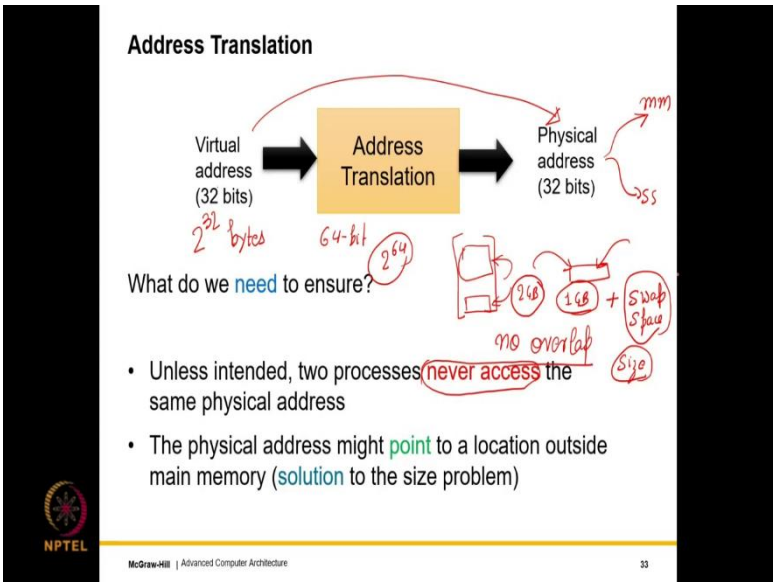
(Refer Slide Time: 27:08)



So, what solves both the problems? Virtual memory. So, the key idea is that the processor generates addresses which are virtual addresses. So, virtual addresses preserves the abstraction. Virtual addresses preserves the abstraction that the process owns the entire memory map at least from 0 to 3 GB. It entirely owns.

Then we have a translation module which has traditionally we have been referring to it as a memory management unit. The translation module generates physical addresses. And these physical addresses are generated in such a way that we solve both the problems of the overlap and the size.

(Refer Slide Time: 28:13)



So, what am I saying? What I am saying is that the program the compiler and the processor they all see the virtual address which is 32 bits. Then the addresses go through a process of translation where the virtual address is converted to a physical address. This process of translation ensures two things. What does it ensure? It ensures that unless intended. Sometimes we may intend to, but unless intended two processes never access the same physical address.

So, the same physical address is never accessed by two processes right unless we intend to do. So, and most of the time we do not intend to do. So, there is no overlap right. Furthermore, we might be limited by the size of the main memory. In the sense the virtual address space is essentially $(2)^{32}$ in this case $(2)^{32}$ bytes.

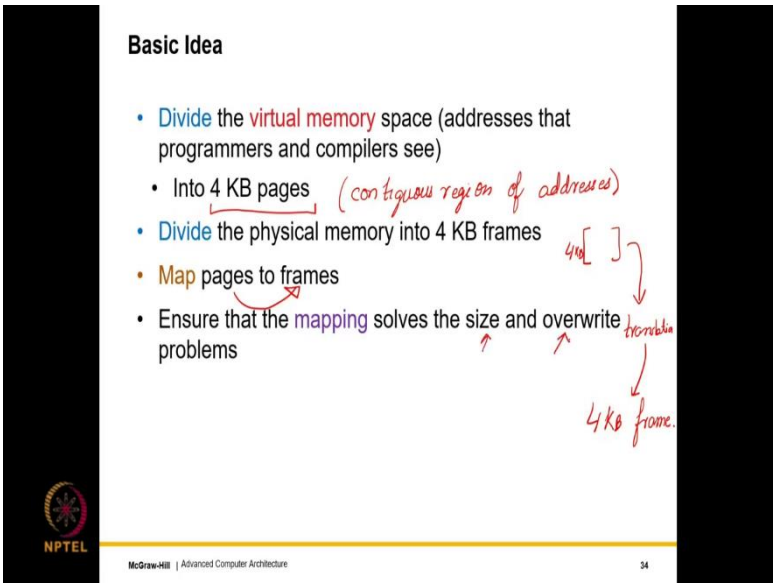
If you are using a 64 bit memory system the virtual address space is $(2)^{64}$ bytes. So, the virtual address space is huge. So, within this of course, we in our entire memory map we might be using some parts of it. But let us see even the parts that we are using let us say they sum up to 2 gigabytes and our actual memory is 1 gigabyte.

Then clearly our program will not run because we have said that we will never have misses in main memory. So, let us slightly relax this assumption. So, let us also use other forms of storage which are much slower much more inefficient such as the hard disk to expand the size of our physical address space or the space of physical locations.

So, this will ensure that even if we need 2 gigabytes of memory, the system will not stop our program our program can still run. It will use the off chip main memory which is 1 gigabyte + an additional region on the hard disk called the swap space. So, that will provide the additional space to store the remaining 1 gigabyte.

So, this solves the size problem for us. And the translation mechanism will essentially say if the address is there in the main memory or is the address there in the swap space. So, the process of translation is important to solve both the problems, the overlap and the size problem.

(Refer Slide Time: 30:57)

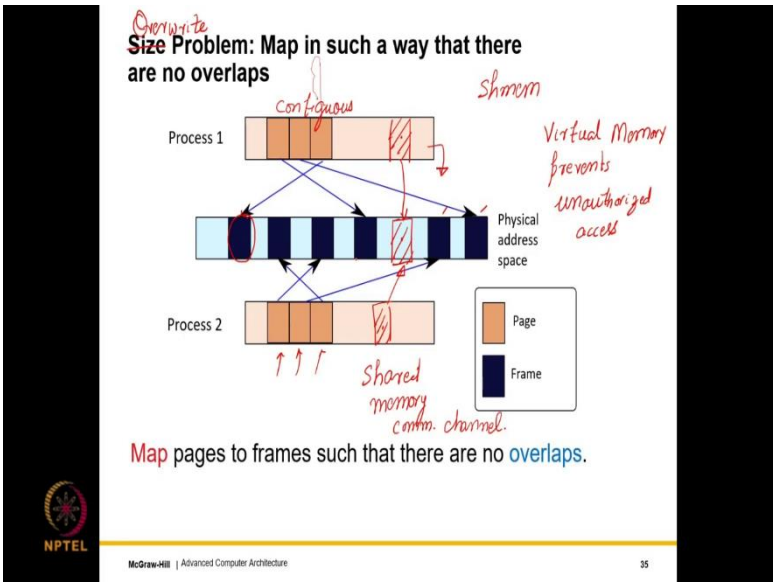


So, here is how we propose to do it. We will divide the virtual memory space. So, what is virtual memory is the addresses that the programmers and the compiler see. So, it is the addresses that your program works with the addresses we have been working with till this time. So, that is the virtual memory space where we assume that we own the memory map. We divide this into 4 KB pages like 4 KB contiguous region of addresses.

So, we divide this into 4 KB pages. Similarly, we divide the physical memory space into 4 kilobyte frames. So, the entire physical memory space is divided into 4 KB frames and so, virtual memory divided in into pages physical memory into frames. We just need to map pages to frames.

So, basically the program is using a 4 KB region of memory. This via a process of translation is mapped to a 4 KB frame. So, we ensure that the mapping solves the size problem and the overlap problem, overwrite problem. So, the size and overwrite problems are solved.

(Refer Slide Time: 32:40)



How do we do it? So, we map it in such a way that there are no overlaps. So, let us take a look at it. So, let us consider process 1 and process 2. So, of course, we can extend this diagram to multiple processes, but let us just consider two processes for the time being.

So, let us say that in the entire memory map we have mapped these three pages. So, in the process of translation what you can see is these pages can be mapped to any three frames in main memory. So, they need not be contiguous. What is the key point? they are contiguous over here, but this continuity is not seen in the physical address space. In a sense they are mapped to different frames and main memory which are not necessarily close by. Similarly, for process 2, we have three pages. They are mapped to three different frames in the physical address space. These are not necessarily close by. So, they are spread across the entire physical address space.

So, they are spread across everywhere and so, it does not matter. It does not matter where they actually are. What matters is that they are essentially mapped and also the what also matters is at the same frame is not mapped to two pages across processes because we have no intention of creating an overlap.

So, that is the reason what you can see over here is that the same frame is not mapped to two pages. So, there is no overlap which also means that even if one process has a very malicious intent and it will even try to access the data of another process which can be a password or a credit card number, the virtual memory mechanism will prevent it from doing so.

So, it in a sense enforces the security of the system as well. So, virtual memory prevents pretty much unauthorised access. So, all that the program can do is that it can access any virtual address in its virtual memory space or in its memory map. But the final mapping from virtual to physical is in the hands of the processor and the processor the processor and the OS also. So, we will see how the OS comes into play.

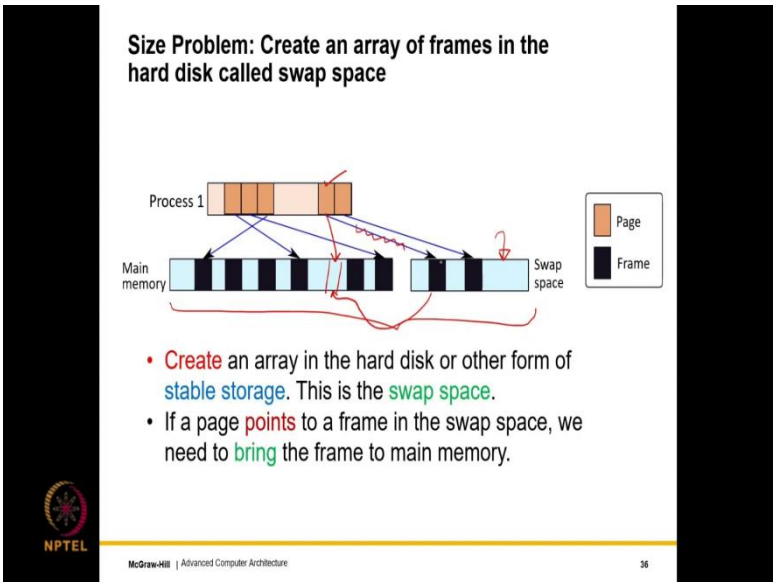
But the processor will essentially guarantee that the same physical frame is never mapped to two processes. So, there is no way that one process can see another processes data. So, thus the virtual memory mechanism enforces it. Also sometimes we might desire to create a overlap.

In the sense sometimes we might want to have interprocess communication, IPC. And is another definition of IPC, not the one that we are using. So, in that case, well there is something called the shmem call on Linux the shared memory call. If we can map one virtual page of one process and another virtual page of another process to the same physical frame so, in this case if the process writes to any byte over here, this will reflect over here which is going to also reflect over here. So, they are they become the same.

So, this is an example where we deliberately create an overlap to create a shared memory communication channel. So, we deliberately create an overlap to create a shared memory communication channel and this is not a security flaw because this is being created with the explicit consent of both processes.

But, in general we do not avoid; we in general we avoid creating such an overlap. There again the reason being security. And also the fact that processes will not be able to run correctly if let us say their values are overwritten by other processes.

(Refer Slide Time: 37:14)



Let us now solve the size problem. So, what we do is that along with the physical main memory. We create another region in the hard disk or any other form of stable storage can be a flash drive as well. So, there we take a region such that the combined size of the main memory and the swap space is equal to the actual memory requirement of all the processes. This ensures that even if we are short of actual main memory which is made of d RAM cells we will still have enough space.

But of course, the swap space is much slower and it is significantly slower rather. So, we need to do some optimizations. Nevertheless what the mapping scheme will do for us is that given a virtual page given a page in the virtual address space, it will map it to either a frame in main memory or to a frame in the swap space. And the mapping will indicate if the frame is in the main memory or in the swap space.

Secondly before using the frame, so, before accessing bytes in the frame we need to bring the frame from the swap space to main memory and put it somewhere and then update the mapping. So, let us say for example, we want to access this page. Then we would learn that this page is currently there in the swap space. So, what we need to do is that we need to bring in the frame.

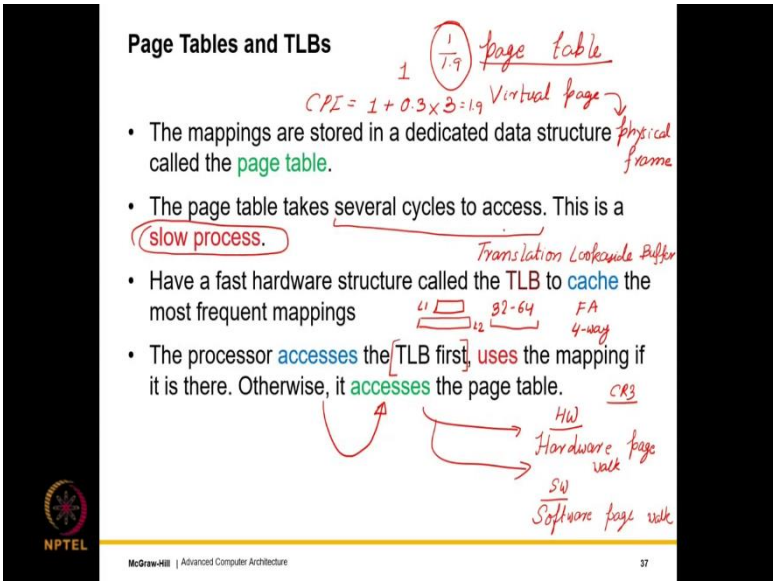
We meaning what the operating system needs to do in collaboration with the hardware is bring in the frame from the swap space to main memory and then update the mappings

such that this points to this position. Of course, if we do not have free frames in memory. It is similar to not having freeways in a set.

So, in that case what is needed is that we need to again take one frame from main memory put it in swap space like do a page replacement create space for a frame and bring a frame in. So, that needs to be done and this process of replacement which is called page replacement is done by the operating system. So, this is also where the operating system plays a key role as a memory manager.

But otherwise the operating systems role is not required if the mapping if the frame is coming from main memory, but to manage the swap space the operating system is required.

(Refer Slide Time: 40:10)



So, where are these mappings stored? The mappings are stored in a dedicated data structure called a page table. So, what is the page table? The page table given a virtual page, it just maps it to a physical frame alright. So, unfortunately the page table is a rather elaborate data structure and the process of doing the mapping takes several cycles.

So, we can reason at it from common sense that the CPI will actually become very low if every memory access actually takes several cycles to access the page table and do the translation. This would indeed be a very slow process and this slow process will reduce our CPI significantly. So, assume that our program has no dependencies then in an in order

pipeline the IPC will be 1, but in this case sorry, I stand corrected I had said the CPI will decrease it will actually increase.

So, what will decreases IPC. So, if we do a little bit of math then we can see that if let us say a third of the instructions are memory instructions. So, we can use our CPI formula. So, CPI will be 1+ the fraction of memory instructions, so which is 0.3. Let us again make an assumption that all the accesses hit in the L1 cache.

So, if I just consider the overhead that is required for translation and let us assume that it takes 3 cycles to translate. So, then the CPI actually becomes 1.9. So, the IPC will then become one by 1.9 which is roughly half. So, this is a massive hit in IPC, this is not something that we can tolerate right.

So, what we have is that we use the same logic as the cache. We create a small cache for the mappings. The cache for the mappings is known as the TLB. I should have expanded. It is called the translation look aside buffer. It is basically a cache for the most frequent mappings. So, because of temporal locality or TLB is extremely effective. The TLB hit rates are very high the order of 99.5%.

So, we have a extremely fast hardware structure called the TLB to cache the most frequent mappings and the TLBs are typically very very fast. So, they typically have 32 to 64 entries. They often are they are either fully associative or they are 4 way set associative, 4 to 8 way set associative.

In a lot of modern designs they also have a two level TLB in a sense they have a fast L1 level TLB and then a slightly slower L2 level TLB. Sometimes also to further optimize we have a separate instruction TLB called an ITLB and a separate data TLB called a DTLB.

So, regardless of what is the configuration the processor accesses the TLB first and the TLB access is typically very fast. It is typically a fraction of a cycle or maybe 1 cycle at the most. It uses the mapping that is there in the TLB to translate the virtual to the physical address. If it does not find if it does not find the entry in the TLB if there is a miss in the TLB then the processor accesses the page table.

So, the question is how do you access the page table. This itself is a rather elaborate operation. So, there are two so, there are broadly speaking there are two methods. So,

accessing the page table to find the mapping and populating the TLB that is what we are looking at because even if the mapping is there in the page table the processor cannot use it unless it comes to the TLB.

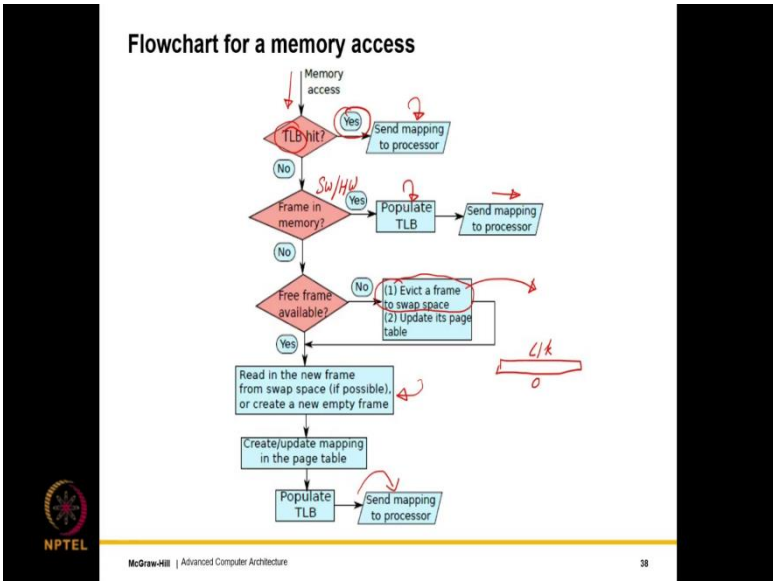
So, now accessing the page table can either be done in software or in hardware. If it is done in hardware which is typically the case in most Intel processors, this is known as a hardware page work. So, Intel processors provide a register a machine specific register called an MSR register. This the name of this register is typically CR 3. So, here we store the base address of the page table.

So, small routine, which is stored in firmware in a set of instructions that the hardware already has. It uses it to access the CR 3 register the page table register. And it automatically walks the page table, traverses the page table, finds the mapping and then updates the TLB. So, this is a hardware page walk and this is efficient in many other systems if you do not find an entry in TLB and interrupt is raised then the operating system does a software page walk. And there the operating system this is a privileged operation. So, the operating system then finds the mapping and populates the TLB.

So, regardless of how it is done? It can be done either by the hardware or it can be done by software. Does not matter how it is done, but ultimately the mapping has to come from the page table. If there is no mapping one such mapping has to be created and then the TLB has to be populated. And the TLB the fast structure, it is the cache for the page table, it is an entirely hardware structure. It typically contains 32 to 64 entries at least the level 1 TLB.

And it is normally a 4 way set associative structure, it can be a fully associative structure as well given the fact that its size is small and it has few entries not much not many. A fully associative cache is good enough for storing the entries of the TLB.

(Refer Slide Time: 46:39)



So, this is a final flowchart for a memory access. This figure is extremely important. So, we should understand it. So, here if we have a memory access right over here, if we have a TLB hit then there is no problem. We just send the mapping to the processor and there is absolutely no problem at all.

If we do not have a TLB hit if we in a if there is no TLB hit then what we do is we try to see if the frame is in memory or not. So, how do we do that? Well, we walk the page table and we see whether the frame is there in the memory. So, the walking the page table is done either by software or hardware and if the frame exists in main memory. Then the software or the hardware populates the TLB and sends the mapping to the processor.

Otherwise, let us say that we are trying to allocate new memory. So, different operating systems will manage this differently. So, some operating systems will simply not allow you to access an unmapped frame and some might say that look, if you are accessing an unmapped frame that is fine, but then the first time you access we will create space for the frame.

So, different policies can be there, but what is important is that if new space needs to be created we need to see if space is there or not. So, is a free frame available, if it is not there then we have a page replacement policy which follows the same methods of cache replacement FIFI, LRU and so on.

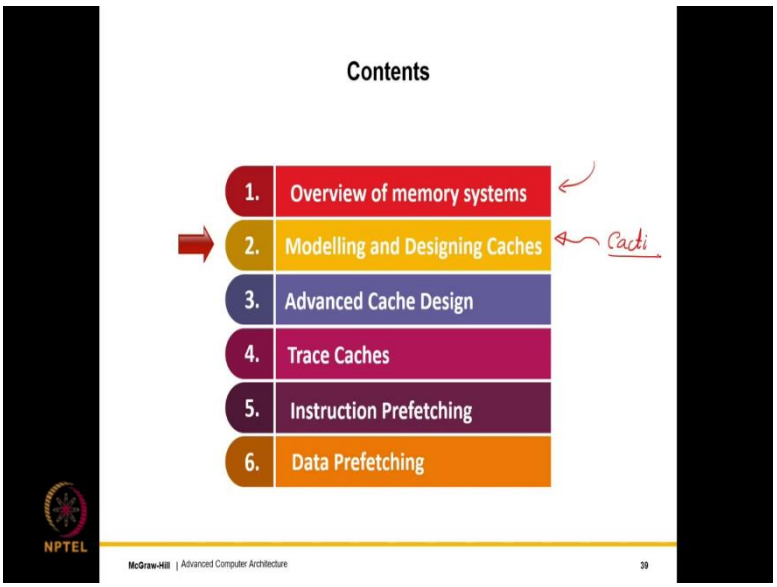
So, we evict a frame to the swap space. We update its page table entry of that to now indicate right. So, we need to update the TLBs and the page table so indicate that this frame no longer exists in main memory, but it has been sent to swap space. Now that a free frame is available we read in the contents of the new frame from the swap space if possible or we create a new empty frame. If a new empty frame is created, we typically 0 out the entire region. Why do we fill the entire region with 0s?

Well, the reason is that it could have contained some previous data written by some other process and that data could be a password or a credit card number which clearly we want to delete. So, we typically 0 out the region. We create a new frame. Create a mapping in the page table, populate the TLB and send the mapping to the processor.

So, this is again the typical flowchart the typical way that we process pages and frames and the most important concept here is the TLB and the page table. So, a book on so, my previous book on computer architecture has a detailed description of how the page table is designed. So, it is typically a two level structure. So, it is I mean two or a multi level structure because the point is that we cannot have an entry for every single virtual page it will be huge.

So, we will use some patterns to optimize the design and also a book on operating systems will also have the design of the page table, but the page table is the key mechanism by which the operating system fulfils its role as a memory manager.

(Refer Slide Time: 50:25)



So, we are now done with a basic overview of memory systems. So, we are in a good position now. So, we have picked up the basics then in the next part we will discuss the Cacti tool. In the Cacti tool we will look at the electrical aspects of designing cache. So, all the details of the SRAM cells, the CAM cells and so on we will look at all of those in the design of the Cacti tool.