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

Lecture – 33
The Memory Systems Part-IV

(Refer Slide Time: 00:25)

So, up till now we have assumed that a program perceives the entire memory system to be its own. So, if every program on 32 bits machines you know; how large is a memory system that it can potentially address well. So, if the addressing is 32 bits we can address all the bits with in all the bytes. So, the address is 0 to 2 to the power 32 minus 1. So, that has a total number of bytes which we can address, because mind you one location contains only 1 byte. So, total we can address 2 raise to the power 32 bytes, which is 4 times to raise to the power 30 which is. So, 2 to the power 32 is 1 gigabyte. So, this is 4 gigabytes.

So, 4 gigabytes is a total amount of memory space that we can address and within this, we can access any location at will. So, this is the assumption that we make all right that the entire memory system is hours. We also need to take multiple programs into account. So, enough for example, at this point may be 100 programs are running on my machine. So, my CPU runs program A for some time, then switches to program B and then to program c. So, in the sense, this is called a time shared system where we run one

program for sometime one more program for some more times, and one more programs for, you know some more time. So, the question is that, do they corrupt each other's data. the answer is no. So, you know; for example, if I have programs a and b and let us say I run program A for one second, and then I run program B for one second, does a deliberately write into the memory space of b, and corrupted it is data. the answer is no, and does b deliberately write to as data deliberately or in a non deliberately, you know just like that in advertently does b write to as space, the answer is no.

So, somehow we are able to achieve a partition of the memory regions used by a and b, but again how is this possible, because a assumes that the entire memory is its own and b assumes the entire memory space is its own. So, a raise to write to, let us say location 20 and b writes to location 20, then they will end up over writing each other's data, which is exactly what we do not want. So, this is also called the overlap problem, and this is something that we want to solve that we have ensure.

So, you know this Aim of here is a good that both a and b actually should think that they are the owners of the entire memory system; however, in practice we need to ensure that they do not over write each other's data. So, you know this is Aim 1 for us. So, let us call it Aim 1, and then there is an Aim 2. So, you know if we consider physical memory. So, what is physical memory is a total amount of physical storage that we have right, in the main memory that is the physical memory.

So, if we consider a memory system that has less than 4 gigabytes of physical memory. So, let say it has 2 gigabytes of physical memory right. We need to ensure that all our programs still run. So, if a program accesses a location which let us say after 3 gigabytes, it should still run on this system. So, this is something that has to be ensured that even if the size of our physical memory. Physical memory is a actual amount of you know memory that we had right, you know d in terms of d ram chips and so on. So, this is actual transistors that we have. So, if this is 2 gigabytes which is less than or virtual memory space which is 4 gigabytes, which is the entire space that can be addressed. We need to find a way ensuring that programs work.

(Refer Slide Time: 04:50)

Let us thus **define** two concepts ...

* Physical Memory
  * Refers to the actual set of physical memory locations contained in the main memory and the caches.
* Virtual Memory
  * The memory space **assumed** by a program.
  * Contiguous, without limits.

So, this is also called the size problem. So, two problems size and overlap. So, let me just you know redefine these two concepts. Physical memory refers to the actual set of physical memory locations that are contained within the main memory right. So, the main memory is the set of memory chips, which contain d ram cells and. So, this is the actual physical memory; and of course, we have the caches as well, but if we consider in the inclusive cache hierarchy, then essentially the size of the physical memory is the size of the main memory, and this can be you know anything it depends of the monto hardware that we have invested.

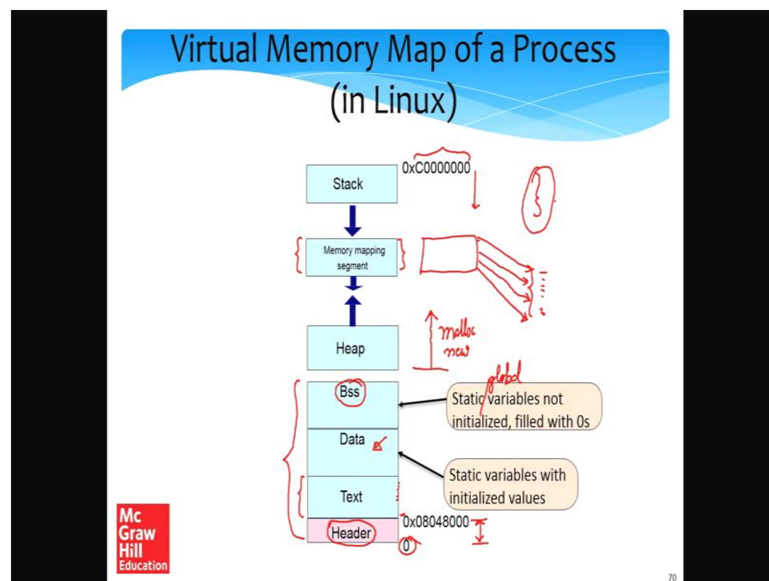
But virtual memory is what the program think it helps. So, physical memory what you actually have, but virtual memory is the amount of memory that a given program or a given programmer thinks that you have. So, this memory space assumed by a program, which is assumed to be contiguous and without limits, and let us if a memory addressed takes n bits then the total virtual memory sizes 2 raise to the power n. in other words if let us say that it has a 32 bit addressing that we are using, then the total virtual memory sizes 2 raise to the power 32 bytes or 4 gigabytes.

Similarly, if we were using 64 bites, then the total virtual memory sizes to raise the power 64 bytes which is huge. So, which is essentially 16 times; yes, roughly 16 times 10 to the power 18 bytes, which is massive all right. So, essentially the idea here is, that you know we have defining two concepts, and the concepts are again different, physical

memory is the actual amount of storage, and virtual memory is the amount of storage that you think that you have right. So, both are different right one is in ideal dream world the virtual world, and the other is the actual world, where you have actual storage devices and the two problems that we need to solve its space and overlap.

So, overlap means a two processes a and b do not end up, you know over writing each other's data, even though the physical memory shared between them, and the space problem is that, you know your physical memory can actually be less than the amount of virtual memory that you have. In that case your program should still work. So, we need to find that how to enable that.

(Refer Slide Time: 07:27)



So, let us look at a process you know then virtual memory map of a process in a Linux operating system. So, we recall that a process is a running instance of a program right. So, all programs, so this is a ILF format. So, all programs actually start with the virtual address 0. So, from 0 till a certain address, from 0 till the certain address we have the header that the header is basically. So, 8 records, the type of the program, the type of the encoding. So, it is lot of meta information in there then we have the text section of the program, which is actually the instructions of the program. So, they pretty much start from this virtual address; after having all the instructions of the program with the data section which consist of static variables and global variables in static oblique global variables with initialized values.

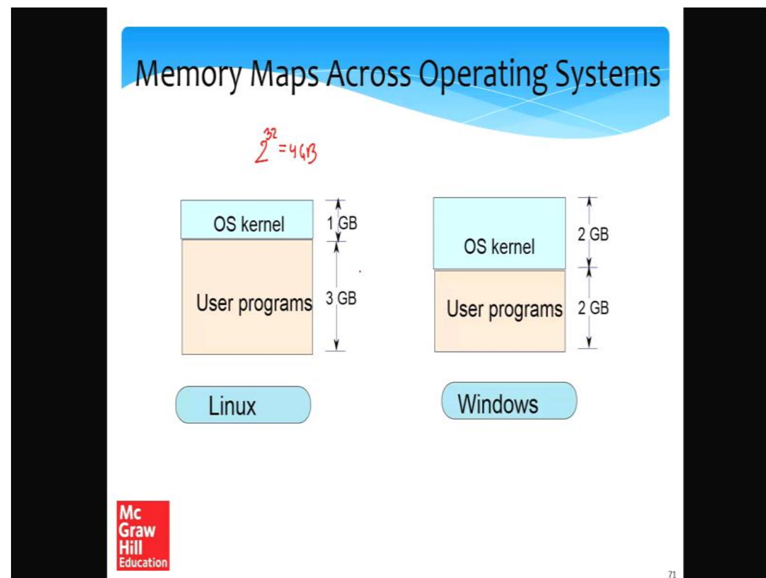
So, let us say in a beginning of a program, I define in text equals to 10. Since this data is initialized, it will come to the data section. We then have a basic section, which also contains in a static oblique global variables same thing, which are not initialized. So, that is the reason they are all filled with zeros right. So, anything that is not initialized it is filled with zeros. So, that is the basic section. So, this is how one part of the virtually memory map is divided, and if I look at the top. So, at a very high address zero x c e and then seven zeros is when the stack starts right, and since the stack is downward growing, it grows to it is lower addresses. and similarly after the basic section, the heat starts we call that the heat is memory area of where we put in dynamically allocated data, like the data that is allocated with calls like melloc in C and mu in C++ and java. So, all the data will go on the heat.

In the middle we have a segment called a memory mapping segment, this is fairly complicated; so I will not be discussing it, but if you want me to give a 1 line summary. The one line summary is as follows that, let us if I open a file or something right file or a disk file on a hard disk. So, which we are not discussed, what is the hard disk. So, sort of you know ask readers to take this of the pinch of salt, what we can do it. We can assign a memory region that corresponds to something that we are, you know trying to open like a file.

So, if I have a certain movie file then in the movie file stored somewhere else. I can assume that this region has a one to one correspondence with the bytes in the movie file. This is all that I can say at this moment I really do not want to go deeper; however, after getting a understanding a virtual memory, we will be in a position to discuss this in some more depth, in chapter number twelve, the last chapter. We will discuss memory mapping, but let us leave this for the time being.

So, basically what happens, every single process has a roughly similar memory map, where we start from address 0. First at the header then the text, then the sections to store global oblique static variables and we have spaces for the heap and the stack. So, something that can be deduced, very easily, is that most of these. So, let us say the stack the addresses will be common across processes. So, how come they will not over right each other? So, we will discuss solutions to this problem.

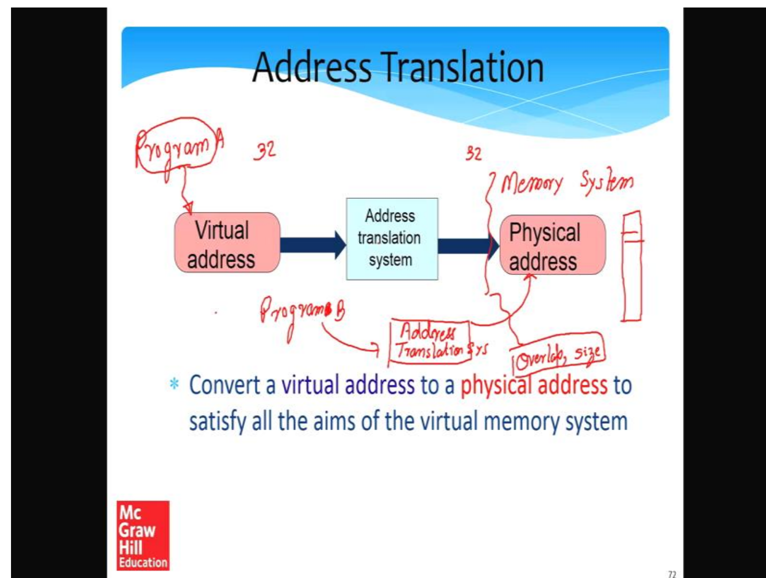
(Refer Slide Time: 11:30)



But before we do that, let us take a look at some memory maps across operating system. So, 4 basically if we consider 32 bits, the total virtual memory size is 2 to the power 32 which is 4 gigabytes. So, what the Linux operating system will do is that it will give 3 gigabytes to user programs typical programs that run, and it will keep 1 gigabyte for the operating system itself.

This is a first time that we are hearing the word operating system. So, the operating system is a special program whose job is to manage other programs, or the windows operating system. What it does is that its splits, it does a 50-50 split. So, it gives user programs to 2 gigabytes, in the 32 bit version, and it gives the operating system called OS kernel to gigabytes.

(Refer Slide Time: 12:29)



So, there are minor differences. So, let us now look at the main idea of what virtual address is, right of what virtual memory is. So, if I consider a program right, all the addresses inside a program are virtual addresses, where I assume that the entire memory space belongs to me, and I can access. And the way I split the memory space, is this way between header text data BSS heap, and stack. But mind you here again the assumption, that is being made is that the entire spaces available to one process. So, this is the virtual addresses that we use; however, this is not the address that goes to the memory system. So, if I consider this as the memory system, the address that goes to the memory system is actually something else it is the physical address.

So, between the virtual address at program users a compiler uses and the physical address that is sent to the memory system we need to have an address translation system. So, this is called with different names, it is also called the memory management unit. They address the virtual to physical address translator. So, call it by any names. So, any name we will call it the address translation system, it basically you know it converts one in our running example 132, but bit address into another 32, but address and it sends it to the memory system.

Now, if I consider one more program. So, let say this is program A, and this is program B I am sorry. So, program B will have one more address translation system of it is own, which is again in a different and this address translation system, would again yield a

physical address, because mind you the physical address is not different. The physical address is the same, because it corresponds to an actual location right in the caches, or in the main memory. So, anything that goes anything inside the processor is a virtual address. Anything that goes out of the processor to the memory system including the one cache right, is the physical address, and for every program that is running. It will have it is address translation system.

We will discuss this in great detail which converts a virtual address that the program uses to an actual physical address, and the physical address is the same for all programs, because it is corresponding to an actual location, and the virtual address is do not matter and also the Aim of the address translation system, is to solve both of our problems, which are essentially the overlap problem ensuring that a and b do not over write each other's data, problem number one. And to ensure that the size problem which is that if you know the physical memory is less than the virtual memory right. We do not have enough physical memory then how to take care of the situations.

These are the two problems that the address translation system must solve for us, must solve for us. And so that is the reason we have virtual memory on one side of the translation system, which is what the program thinks it has. The entire memory space, one large contiguous memory space, and the physical address which is on the other side, both are separated by a translation system.

(Refer Slide Time: 16:19)

Pages and Frames

* Divide the virtual address space into chunks of 4 kB → page
* Divide the physical address space into chunks of 4 kB → frame
* Map pages to frames
  * Insight: If a page/frame size is large, most of it may remain unused
  * If the page/frame size is very small, the overhead of mapping will be very high

So, to describe the translation system, it is not possible to do in one slide, it will take a multiple slides. So, let us use the multiple slides and describe the translation system in some amount of detail. So, let us divide the virtual address space that we have write the total memory space, that a program thinks it as into chunks of in a contiguous consecutive chunks, the same way we had blocks in a cash. We divide the address space into chunks of 4 kilo bytes, and each such chunk is called a page all right. So, each such chunk of consecutive 4 kilo bytes it is called a page. And similarly we divide the physical address space into again chunks of contiguous chunks of 4 kilo bytes this is called a frame.

So, mind you the virtual address space and the physical address space can have different sizes for example, it is very much possible in modern processors, where the virtual address to be a 48 bit quantity, and may be the physical address to be a 36 bit quantity; that is perfectly fine your address translation system will map a 48, but quantity to a 36 bit quantity, no problem in that all that we are saying is that. Let us take this address same way we made cache blocks right. The same way we had atomic indivisible cache blocks, let us make a larger block, let us call it a page when we are creating blocks out of the virtual address space, and let us call it a frame, when we are creating blocks out of the physical address space.

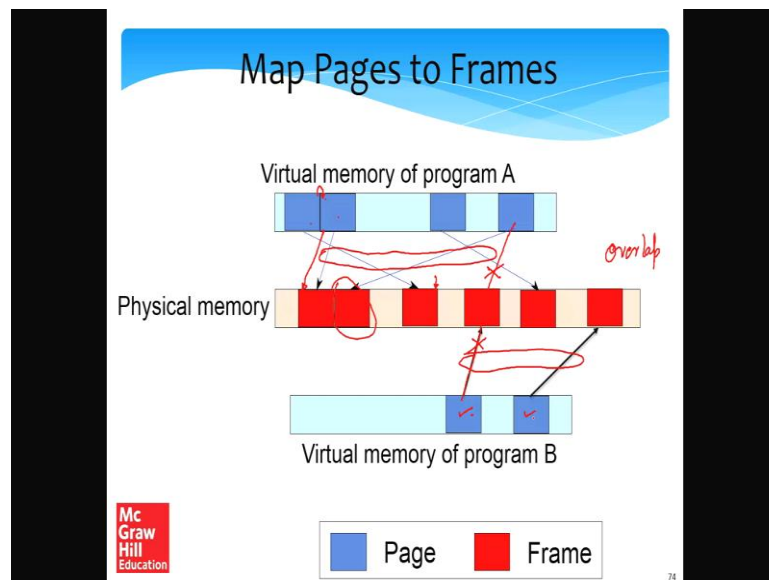
So, the job of the translation system is to map a page to a frame. So, let us consider this number in some more detail 4 kilo bytes right, 4 kilo bytes is 4 is 2 square, and a kilo is 2 to the power 1024. We are approximating to a 1000. So, this is 2 to the power 12 bytes and. So, it will take 12 bits to specify the address of a byte within a page. So, what we can do is, given an address we can, let us say given 32 bit address, what we can do. So, let us consider 32 bits, because it is a running example, but we can easily extend it to other memory systems. So, we will take the bottom 12 bits which is essentially the offset of a byte within the page, and discard them. The remaining 20 bits are the page address. These 20 bits need to be mapped to another physical page, which is a frame.

So, we will discuss about this in some amount of greater detail in the next slides, must let look at the mean inside. Well the mean insight is that you know why are we doing why are we making that blocks. Well the reason we make the blocks is as follows that it will reduce our translation overhead. So, instead of you know doing a translation for every byte, we can actually translate it at a much higher rate. So, it will save us some efforts.

So, first is as a page is really large. Let us say it has like 100 megabytes, then you know most of the page will actually remain unused, and since we will not divide a. So, page size and frame size mind you have to be the same. So, since we are not dividing a physical page; since you are not dividing a page or a frame into multiple sub paths, if we have a really large page, we might end up wasting memory.

Similarly, the page or frame size is very small, you know let us assume that it is only like 8 bytes, and then we will have the overhead of mapping will be extremely high. So, we will have to map very frequently. So, that is again a bad idea. So, 4 kilo bytes has been experimentally been found to be a good choice. So, we take 4 kilo bytes chunks, and for each of this, we map each page to a frame. So, let me discuss in some more detail.

(Refer Slide Time: 20:40)



So, the mapping process is as follows. Let us look at diagram of course, not drawn to scale of the virtual memory regions that a program A, process A uses. So, let us assume that the entire blue is the entire virtual memory region of process a. So, mind you the diagrams are not draw to scale. So, out of this light blue, is the regions that are not used right. So, basically they are not accessed and the darker blue is the color of the regions that are accessed.

So, in these regions we divide the regions into 4 kilo byte pages. So, each such square box is a 4 kilo byte page right. So, this is a 4 kilo bytes page. So, each page is mapped to. So, basically if you consider process a, it sees a contiguous virtual memory, where the

virtual memory occupies the entire memory space, and it does not need to see any other view, but what happens in the middle in of this sort of the secret saws is that, there is a mapping layer that the program does not get to see, and you know that is the secret saws over here.

So, essentially every page is mapped to a frame. So, this is the crux of the idea every page is mapped to a frame, but mind you this page is mapped to this frame, this page is mapped to this frame, this page is mapped to this frame, and this page is mapped to this this frame. So, they are pretty much map to random frames which are dispersed all over, in physical memory and the frames are definitely not contiguous, and the greatness of this mapping is that the programmer, the process, the compiler can see the virtual memory as one nice contiguous memory, but what happens in reality, is that each of these pages get map to different frames, in the different frames can be there all over right. So, they need not be contiguous. So, this solves most of our problems; the reason is that let us again take a look at you know both of these rectangles.

So, let us consider the virtual memory of program B. So, let us assume that program p addresses memory in two regions where each region fits with in a page. So, again these pages are mapped to different physical frames which of course, totally disconnected. So, by the interesting way in which the overlap problems is being solved is like this that if you see we are not allocating a physical frame to pages belonging to two separate processors right. So, for example, this physical frame is only you know allocated to this page and this physical frame have a one to one correspondence.

Similarly, this page and this physical frame have a one to one correspondence. So, we never allot A the same physical frame. So, we never map it to pages in two separate processes this ensures that there is absolutely no overlap right. So, this is exactly what solves the overlap problem, and the way it solves the overlap problem is that the programmers can compilers, can have their own view in reality that is a thin memory mapping layer which is not visible that us a part of the processor secret saws, and this is where the address actually changes, and from the page we page address we go to a frame address the frames can be dispersed all over.

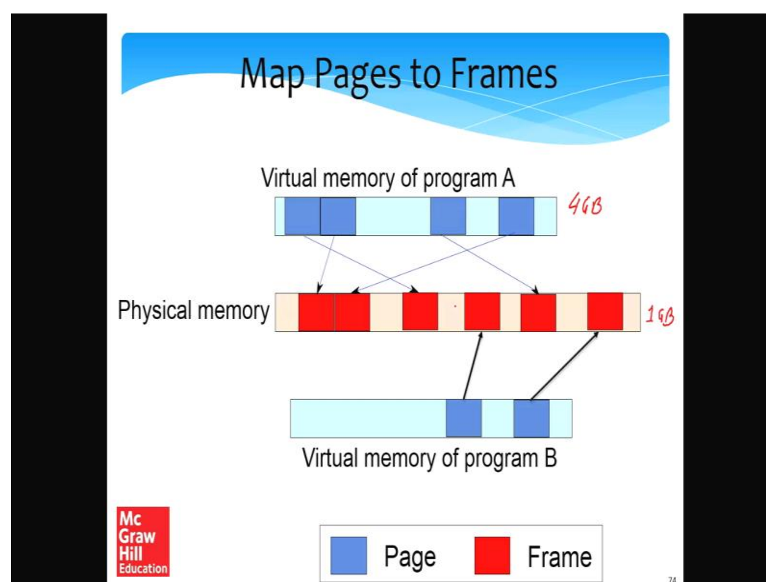
For example, when we move from the last byte of this page to the first byte of this page as for programmer is concerned, we are only if moving ahead by one byte, but what is

actually happening in reality is at this page was mapped to this frame. So, the last byte of this page is a last byte of this frame which is essentially this byte, then again when we move here the mapping changes, this page gets map to this frame. So, basically this byte which is the starting byte of this page actually gets mapped to this byte in a physical frame.

So, the advantage here, is that you know both a as well as b you can see the entire memory space as their own, there is no problem, but since you know they will not be occupying the entire virtual address space which is 4 gigabytes, but they might be occupying let say 10 kilo bytes 100 kilo bytes 1 megabyte we can divide the memory region in two pages per each page is 4 kilo bytes, and because of the special locality you know we will have a nice clustering of data, and these pages can then me mapped to frames and memory the only constraint. Here is that one frame has to be mapped only one page it is. For example, in a mapping this frame to this page and mapping this frame to this page is not allowed right. If it is then purposely as a separate issue, but otherwise it is not allowed.

This ensures that two processes can never share data in the physical memory. So, this solves our overlap problem. So, this slide is very important when just in case we do not understand this I would advise the readers to take a look at this.

(Refer Slide Time: 26:25)

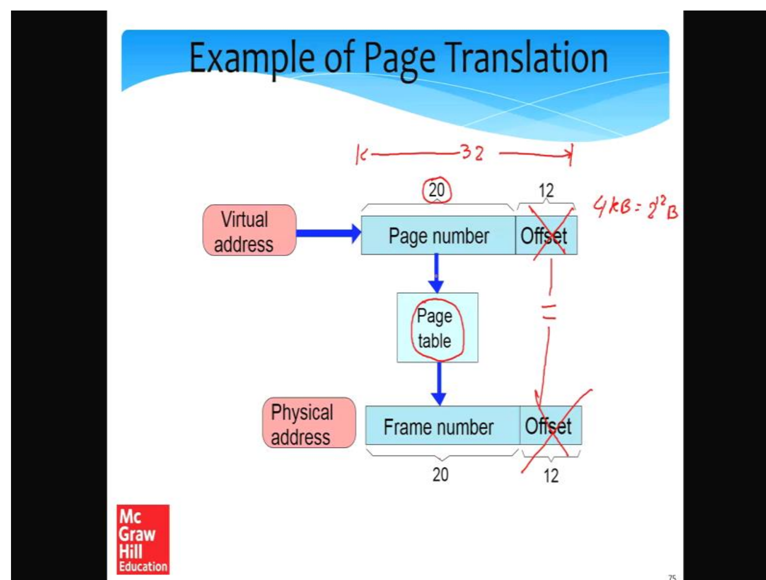


And sort of take the video back and go over this as many times as possible because this diagram is the clucks of virtual memory if this diagram is understood all of virtual memory is understood and if this diagram is not understood pretty much nothing is understood. So, student should make at most effort to understand this diagram.

Let me again summarize what this is doing in two sentences, but to risk loss of interest due to reputation, I will move ahead, what we are trying to say is let processes a and b perceive a nice contiguous memory system in reality, we take the regions at the access you know some block them group them into pages and map each page to a frame in physical memory, that has the same size, and we never share a frame across to separate processes that will solve the overlap problem for us. There are other problems let us take a look at it, so also the size problem can be solved like this, or if the virtual memory size is 4 gigabytes. This is only 1 gigabyte, it makes no difference at all, because in any case we have broken this down into frames, and we just have to map pages here to frames. So, it does not. So, the actual size of this does not matter. It will only matter if the working set of programs a and program B, the amount of memory they require exceeds 1 gigabyte then the entire physical memory will fill up, but that is rare.

So, typically programs do not use that much of memory. So, that is the reason we are safe.

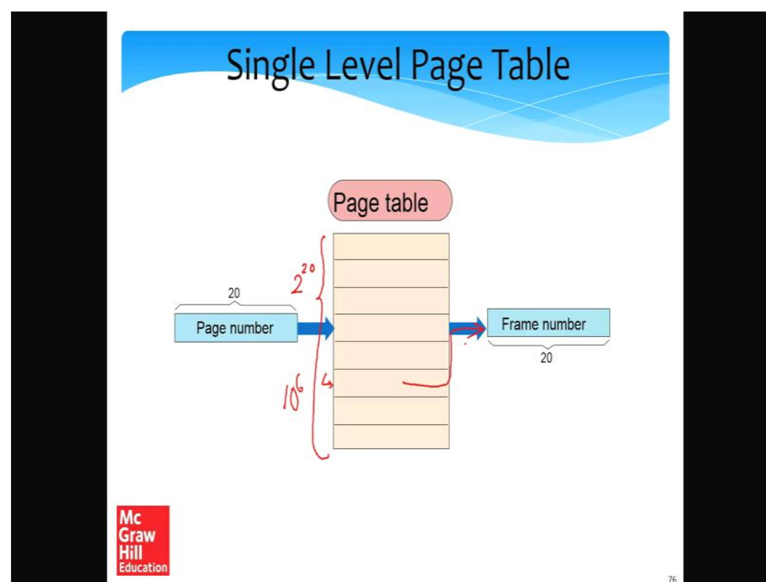
(Refer Slide Time: 28:07)



So, now let us take a look at the details of page translation. So, given a virtual address in a 32 bit addressing scheme since a page is 4 kilo bytes and 4 kilo bytes is 2 raise to the power 12 bytes, the lower 12 bits. We will specify the offset of the byte within the page since same way as the offset of a byte within a block in a cache. So, this part can be ignored right or both this part is the same right it is a same. So, basically you know this is the same as this.

As a result we can ignore these two but what will; however, change is the upper 20 bits which is the page number that will go to the translation system, and what will come out is a 20 bit frame number. So, the translation system is also called the page table, because for every page you know a page number comes in, and a frame number comes out, this is also called the page table. So, in the virtual address 20 bits come in 20 bits of the virtual address, come in and 20 bits of the physical address come out.

(Refer Slide Time: 29:17)



So, let us take a look at a very simple and naïve implementation of a page table. So, this is a, this called a 1 dimensional page table, which is very simple we entered 20 bits, so 20 bits. How many combinations we can have? It will be 2 raise to the power 20, which is one million. So, roughly you know one million combinations, we can have and then each of these entries will have a 20 bit frame number and. So, what we do is that we will like a direct mapped cache. We will access one entry read the frame number from it, and give it as the output.

(Refer Slide Time: 29:59)

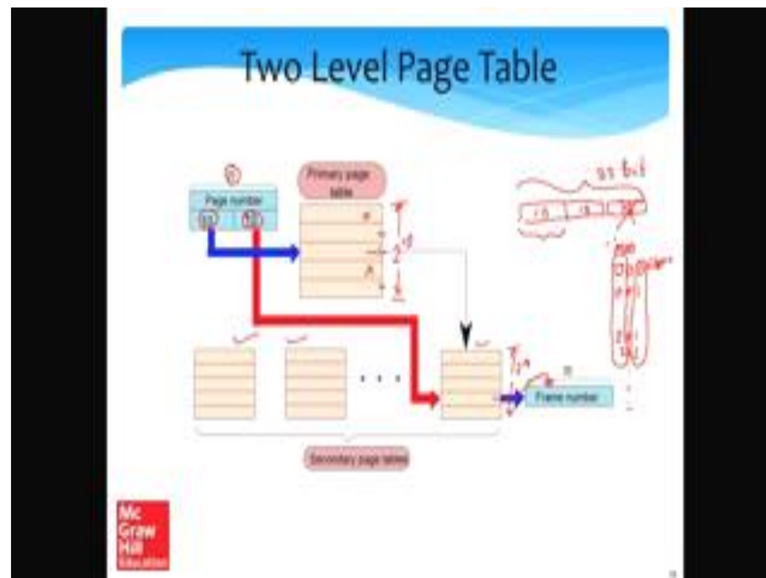
Issues with the Single Level Page Table

- Size of the single level page table
  - Size of an entry (20 bits = 2.5 bytes) *
  - Number of entries (2^{20} = 1 million)
  - Total → 2.5 MB
- For 200 processes (running instances of programs)
  - We spend 500 MB in saving page tables (not acceptable)
- Insight : Most of the virtual address space is empty
  - Most programs do not require that much of memory
  - They require maybe 100 MBs or 200 MBs (most of the time)

So, let us look at the issues with basic single level page table. So, the size of the single level page table can be computed as follows. So, the size of a single entry is 20 bits 2.5 bytes, the number of entries is 2 to the power 20, or one million, the total amount of storage used is 2.5 megabytes. Now the page table is specific to each process. So, if we have 200 processes running instances a program, as is the case that is currently happening on my system. We will spend 500 megabytes of a gigabyte in just storing page table, which is not acceptable. It is a huge amount of memory space; that is being used to store page tables this is just too much.

So, what is the insight? Well the insight is that most of the virtual address space is actually empty for a given program, the virtual address space that it is talking about, is on 4 gigabytes, but out of 4 gigabytes, you know the amount of memory a program might be using might only be 10 megabytes right the rest is empty. So, they will not require that much of memory. So, maybe we can leverage this pattern in some way. So, let us see how it can be leveraged used.

(Refer Slide Time: 31:23)



So, let us consider a two level page table this is called a multilevel page table. So, what we do is that we divide the page number which is a 20 bit quantity into two 10 bit quantities 10 and 10. So, we take the most significant 10 bits right; the upper 10 bits and we use it to address a table, which has 2 raise to the power 10 entries, which is 1024 entries right. So, this is called the primary page table, subsequently the primary page table. So, mind you since a program does not access a lot of pages most of the entries in the primary page table will be empty.

So, I will discuss this in some more detail to it is end of the slide, but let us this can be continue with the discussion, each of this primary page tables points to a secondary page table. So, we initially let us say this entry points to this secondary page table. We take the remaining 10 bits, and use it to address, the secondary page table. So, which again has 2 to the power 10 entries, and each entry of the secondary page tables contains the 20 bit frame number.

This design right where we take 10 bits first access a primary page table. This primary page table contains a pointer to the secondary page table, when we take the next 10 bits and access the secondary page table. This design is called a two level page table, and the claim is that is a far more efficient design.

(Refer Slide Time: 33:06)

Two Level Page Tables - II

- We have a two level set of page tables
  - Primary and secondary page tables
- Not all the entries of the primary page table point to valid secondary page tables
- Each secondary page table → 1024 * 2.5 B = 2.5 KB
  - Maps 4MB of virtual memory
- Insight: Allocate only those many secondary page tables as required.
  - We do not need many secondary page tables due to spatial locality in programs
  - Example: If a program uses 100 MB of virtual memory and needs 25 secondary page tables, we need a total of 2.5KB * 25 = 62.5 KB of space for saving secondary page tables (minimal).

Let us see why, we have two pages tables here. So, there has to be some advantage and let us see if; what is the advantage. So, the first advantage is that you know the way that we do this addressing. So, if we consider a 32 bit memory address and we divide it into 10 bits and 12 bits right. So, anyway the bottom 12 bits correspond to the address of the byte within the page. So, they are not important the upper 10 bits. The most significant 10 bits which are these bits are expected to have the least amount of variance right. So, if I consider you know the. So, let me look at it in some other way, assume that I am counting from zero to thousand. So, then the first number you know in decimal first number is 0 0 0 then 0 0 1, after some point I will reach 2 0 1 2 0 2 this way.

So, felt by see the least amount of variance or the or a larger amount of predictability I see the least amount of variance in the most significant digit right, the most bit byte digit does not matter, and I see the highest amount of variance in the least significant digit, because it changes very frequently right. So, this is where, I see the highest amount of variance right. So, it will put a plus plus here and a minus minus here the same is true for memory addressing. So, typically when I am accessing an array or reading instructions in a program, I would expect the maximum amount of variation, you know in the bits to actually happen for the lower 10 bits of the page number, and I would expect, and I would expect or relatively lower variation to happen in a upper 10 bits look at this counting example.

So, essentially the upper 10 bits remain relatively stable, and you know they change only once every hundred entries, but the lower bits change far more frequently, say if I take this upper bits. So, most likely very few of these you know upper bits will point to valid entries right, and most of the entries here will actually be entry, because no combinations of upper bits would actually point to them. So, this is the good strategy. So, this minimizes the number of secondary page table, that we need to use in this case if are upper bits only point to three separate entries.

So, mind you the primary page table is a direct mapped cache. So, if the upper bits point to only three valid entries. We need to allocate only three secondary page tables right. So, this is allocated on demand it is not that we allocate all 1024 secondary tables that will defeat the purpose. Only the numbers of secondary page tables that are required are allocated. So, in this case only if three of the entries in the primary page table, actually get pointed to vary upper 10 bits more significant, 10 bits, we allocate only three secondary page tables. Then we use the remaining ten bits to address them.

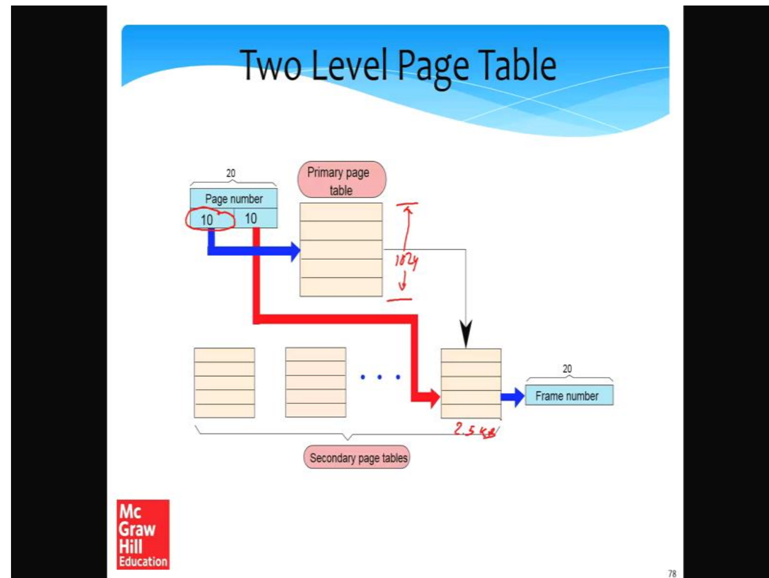
If we want me to do some math, then the math is like this that each secondary page table contains 1024 entries each entry is 20 bits or 2.5 bytes. So, the total memory requirement of a secondary page table is 2.5 kilo bytes. So, 2.5 kilo bytes will roughly map four megabytes of virtual memory, and of course, we will allocate only those many secondary page tables as required. So, let us consider.

So, I have already talked why this is a good idea, and keeping special locality in mind let us consider an example, if a program uses 100 megabytes of virtual memory, and needs 25 secondary page tables, we need a total of 2.5 kilo bytes times 25 or 62.5 kilo bytes for saving secondary page tables, and we are also need a little bit more, you know at max we will also require this much as space for the primary page table. So, the total amount of storage right in a can be this, or can be slightly more, but in any case it is approximately 62.5 kilo bytes, you know plus or minus if you for not minus, but plus a few more kilo bytes for the primary page table, but this is minimal, you know this is absolutely minimal it is you know roughly 60 kilo bytes per process as compared to 2.5 megabytes per process.

So, you know in one you know per process if I consider a one level page table where spending 2.5 megabytes. If I consider a two level page table we are spending 62.5 kilo

bytes. So, this is clearly in a space sufficient by an order of magnitude; that is the reason secondary page table should be used what again is the insight, when the insight is simple the insight is basically.

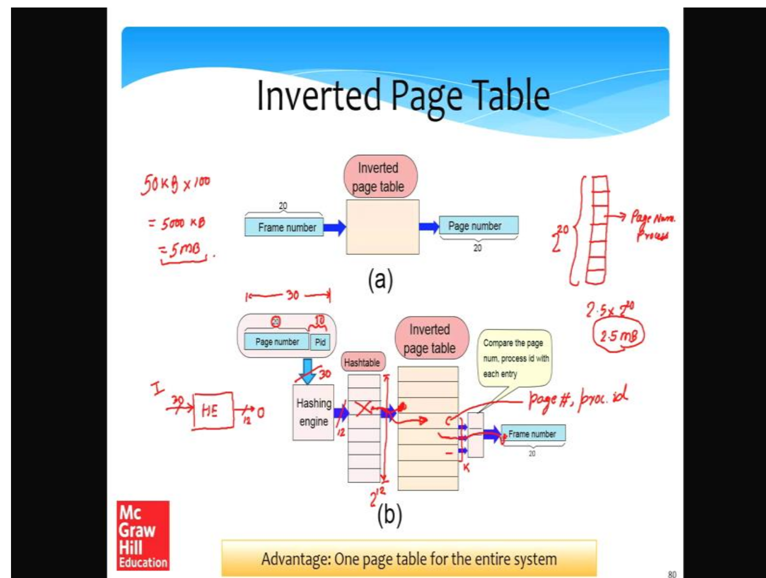
(Refer Slide Time: 38:41)



That we do our addressing in this manner that we divide the address into two parts, and the upper part is expected to show little variance or randomness, you know recall that counting logic. So, very few of these entries would actually be populated and point a valid secondary page tables, and we will not allow allocate a secondary page table for every entry.

So, let us say if you know this has 1024, but may be you know if only 10 secondary page tables are required, we will allocate space for only 10 and each of these secondary page tables is 2.5 kilo bytes. So, the total space utilization is very small very modest. So, that is the reason two level page tables are used in almost all commercial systems.

(Refer Slide Time: 39:31)



Let us now look at a different kind of a page table called an inverted page table the inverted page table per se is not very popular, but it still has a very strong motivation. So, the strong motivation for using an inverted page table are as follows the, let us say the size of one page table is 50 kilo bytes right. So, since the page table is specific to each process, and let us assume that there are hundred processes in the system then the total amount of memory that is used is 5000 kilo bytes or 5 megabytes for certain systems that are memory constraint, but still have a lot of processes, it is kind of difficult to find you know this much of memory to save page tables. So, that is the reason also you know updating.

So, when processes, there is a contact switch one process leaves the CPU and other process starts running on the CPU, the page table needs to be changed. So, given the fact that we have these overheads, it is a better idea at least perceived by some to have an inverted page table, but again you know the two level page table the previous slide in the previous slide, is form a popular, but this is still used notably in some IBM machines.

So, the idea here, is that we have a single page table, where the entries of the page table are essentially they correspond to each physical frame in memory. So, we can logically think of that we are giving a 20 bit frame number is inputs. So, it will have you know somewhere like 2 to the power 20 entries and you know each entry will store the page

number that this actually corresponds to and the page number, and the process if you know which page number belonging to which process.

So, depending upon the math this can technically work out to be a slightly more space efficient solution. So, if we consider let say this particular design, and if you assume that each entry is 20 bit. So, just 2.5 bytes. So, the total size of this is 2.5 bytes times to 2 raise to the power 20 which is 1 mega. So, this is 2.5 megabytes. So, if you have 100 processes, then we see that you know we tries the page table, it might take 5 megabytes if we assume 50 kilo bytes per process, which is kind of reasonable and this takes 2.5 megabytes, and if we have even more processes, then this approach will be more space sufficient and here we have one entry per physical frame for each entry. In each entry we save the page number and the process that this entry is mapping to right, this page belongs to.

So, all of this is finds from a conceptual point of view, but the way we implement it is slightly more complicated. So, we in this case we consider the 20 bit page number and we consider the process. If so, the process is a well, this depends upon the number of processes that we wish to support. So, in a system with thousand processes we will need 10 bits. So, the total size of this in a 20 bits plus 10, the total size of this is 30 bits and. So, we need to take the 30 bits and pass it through a hashing engine.

So, hashing is a very common term in computer science. So, for those who do not know what a hashing engine is. So, let us consider this as a hashing engine, and let us assume that 30 bits are going in. So, what a hashing engine does is that it mangles the 30 bits, and computes another number which as a number of bits, and which is pretty much decided randomly. So, you know this numbers the output has a uniform distribution. So, what this means is that I have, I can think of this as a black box where 30 bits come in and 12 bits come out, which means that for every 30 bits number right, for every 30 bit number, there is a unique mapping to a 12 bit number.

So, basically given a 30 bit input, there is a 12 bit output. So, mind you for every output multiple inputs might be corresponding to it, but for every input only one output corresponds to it. So, if we assume that the input here is 30 bits, and the output here is 12 bits. So, typically hashing engines are considered uniform, you know a good property is called a uniform hashing, which means that all the outputs are equiprobable. So, it has

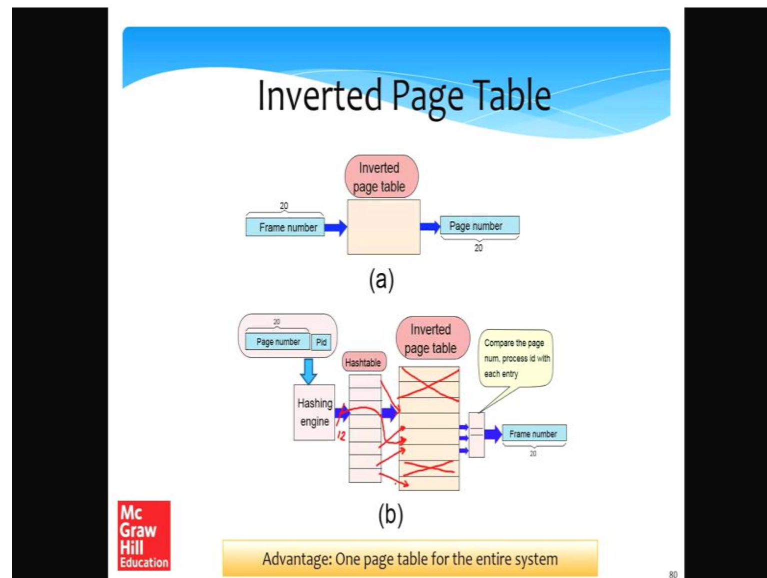
sort of balances the inputs across the output, but that again is a pretty theoretical concept, but what we are concerned with is that 30 bits are coming in and 12 bits are coming out.

So, we use these 12 bits. So, access a table called a hash table. So, since we have only 12 bits. The maximum number of entries that we can have in this table is 2 raise to the power 12 each. So, let us say this map to this entry. So, each entry of the hash table maps to one entry of the inverted page table. So, let us say this entry maps to this entry right. So, it is not a. So, we can discuss more about this mapping, that let us first look at you know go over the figure at least once. So, each entry in this hash table maps to one and only one entry in inverted page table. So, we first compute the hash access, the entry in the hash table, then access the entry you know from here, we access the entry in inverted page table, and some sorry in this example it is not this entry is not mapping to is actually this entry that is mapping to, and in this entry we compare we look at the page number in the process id that is stored in this entry.

So, pretty much every entry in the inverted page table sports the page number, and the process id right. So, we compare over page number in process id with the number; that is stored over here. So, let us assume that you know this it does not match. So, why will it not match? Because multiple inputs can get map to the same 12 bit number to the same output of the hashing engine; as a result they will come here in all of them will be redirected to this entry. So, if there is no match, let us do one thing, let us then search in the next k entries.

So, k can be 1 2 3 10 does not matter. So, in this case it is three. So, let us search in the next k entries, and check each one of them is does the match in the any one of them. Well we are good to go otherwise we will infer that this entry is not there in the page table. So, once let us say this entry matches. So, then this will also score the frame number, and we read out the output frame number.

(Refer Slide Time: 47:24)



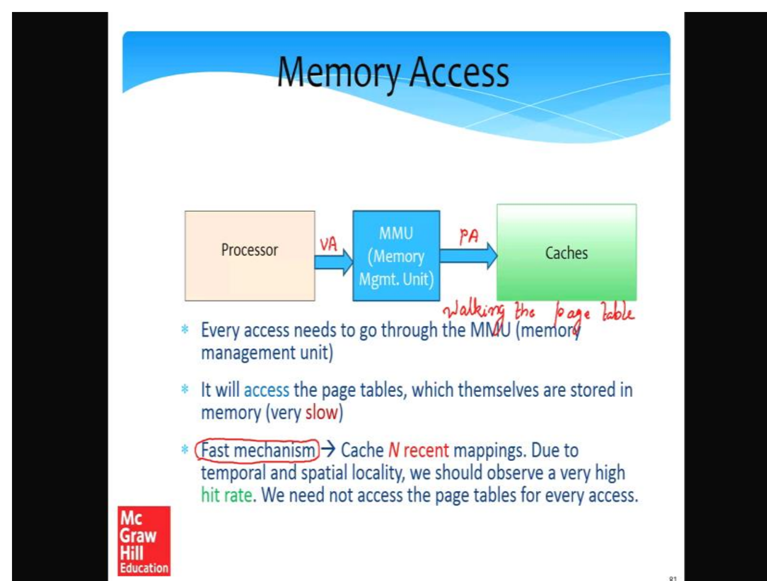
So, let me just over the entire slide once again after erasing the ink. So, the idea here, is that we first construct a number out of the page number, and the current process id which we have assumed to be 10 bits given the 30 bit number, we send it to be hashing engine which mangles the bits and complicates, and we may complicates it and computes the complicated function, where the output will typically be a smaller number; that is may be 12 bits wide right; that is the example that we have been using. We now use these 12 bits to index a separate table called a hash table, and each entry of this hash table points to an entry in the inverted page table.

So, let us say we come to this entry. So, from here let us say this entry points to this entry in the inverted page table. Here we compare the process id, and the page number that has stored over here with our process id, and page number right with which we access this, if there is a match well and good, we rewrite the frame number; that is stored in this entry; otherwise we search over a next k entries, and we read out the frame number. So, the advantage is, we have one page table for the entire system also a you know smart reader would ask what is the necessity of having this hash table, why did not we use the hashing engine directly to access an entry in inverted page table well. So, the answer is like this that may be there are some frames that we do not want to assign to any process. So, may be in a these two frames over here. We do not want to assign to any process we sort or want to reserve them for either the operating system or some other special kind of process.

So, to you know achieve this what we should do or what we need to do is, that we need to add a further level of indirection, where we first actually access the hash table and entries in the hash table will point to entries in the inverted page table. So, for example, if we want that this entry will not be you know map to any page. Then will we basically need to ensure that no entry in the hash table will actually point over here. Had we not add the hash table, then I am since the hashing engine produces these 12 bits which are you know uniform they distributed definitely for some input combinations, we would have map to one of this entries that we do not want to be mapped to, and that would have caused problems right.

So, to avoid this we add an extra table called a hash table. So, the long and short of it to summarize, is that an inverted page table as some benefits in the sense that we do not need a separate page table per process. We can have one page table for the entire system, but; however, accessing it is complicated.

(Refer Slide Time: 50:28)

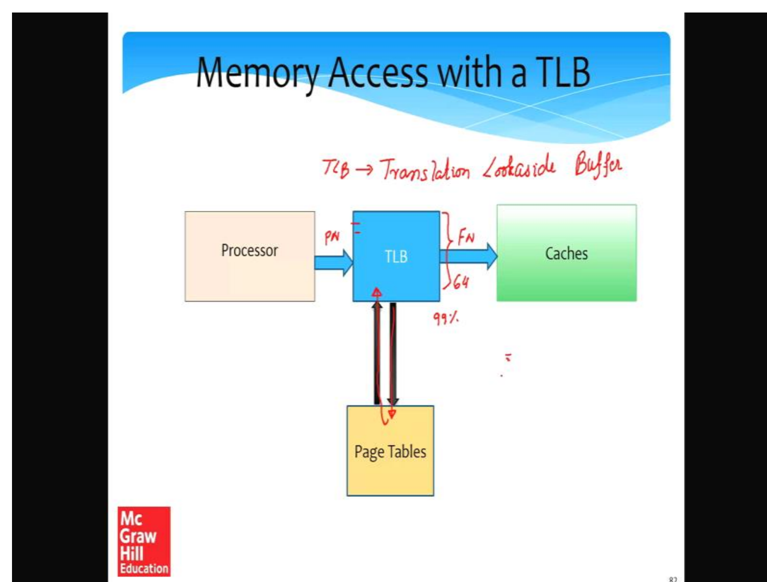


So, let us now summarize the entire memory access process. So, we have the processor that realizes virtual addresses, we have a memory management unit page tables or a part of it that translates the virtual address to physical addresses, and these are send to be caches. So, pretty much every access needs to go to the MMU. So, what we are described up is now way to access the page tables which themselves you know where the page tables are stored. Well the page tables have to be stored somewhere, and they need

to be stored in memory right. We cannot have a separate piece of hardware for them. So, they need to be stored in memory as well and. So, they of course, will be stored in at some point and the physical addresses will be known for never a less, you know this will make this very slow, because for one load instructions will have to walk the page tables. So, a term is walking the page table right, let me just write it.

So, let us you know design a fast mechanism such that we do not have to access the page table for every single memory access. So, what we can do, is that we can have a very small cache that as the n most recent mappings. So, due to temporal and spatial locality, we should observe a very high bit rate in this cache, and we will not need to access the page tables very frequently. So, we add a specialized structure called a TLB. So, TLB is a translation look aside buffer. So, pretty much it stores translation.

(Refer Slide Time: 52:05)



So, TLB is actually very small cache it is a typically a set associative or a fully associative 64 entries cache. So, what comes in is actually the page number and we quickly access the TLB and what comes out is the frame number all right, and since the TLB is a very small structure and it uses the LRU represent scheme right. So, it stores the most you know recently used entries it can satisfy most of the translation requirement memory addresses. So, TLB is typically have a very high hit rate.

The hit rate of TLBs can well exceed you know 97 98 99 percent. So, most of the access actually hit in the TLB. So, TLB miss rate process not that big a deal and this ensures

that we can access or caches very quickly this is, because our translation operation is very fast and for one of those occasional TLB misses where we do not want the page number in a TLB we need to access the page tables and we need to get the translation from the page tables populate at TLB and move on.

So, the TLB can be taught of as a cache for the page table which helps speed up the process of memory address translation and the reason it works is, because of spatial and temporal locality right well spatial well both temporal locality, because we will reuse the same pages over and over again, and spatial locality, because you know is high probability that we will remain within the same page the page is a huge structure is 4 kilo bytes.

(Refer Slide Time: 54:16)

TLB

* TLB (Translation Lookaside Buffer)
  * A fully associative cache
  * Each entry contains a page → frame (mapping)
  * Typically contains 64 entries
  * Very few accesses go to the page table.
* Accesses that go to the page table
  * If there is no mapping, we have a page fault
  * On a page fault, create a mapping, and allocate an empty frame in memory. Update the list of empty frames.

So, the TLB can be a fully associative cache it can be a set associative cache no problem each entry will contain a page to frame mapping a TLB typically contains 64 entries can be 128 also; no problem and accesses say there is no. So, then if there is a TLB miss we go to the page table and it is possible that we are if you are accessing a page for the first time we will not have a mapping for it. So, this is called a page fault.

So, what is a page fault, a page fault is when the page is not available in memory. So, we will see it can happen, because of various reasons. So, one of the reasons can be at least do not find an entry for it in the page tables. So, in this case whenever there is a page fault we need to create a mapping right, allocate an empty frame in memory and also

update the list of empty frame. So, mind you in the memory management unit which can be implemented either in hardware or in software, we need to have a list of empty frames. So, moment we take a frame out of this list we need to update this list pretty much remove that frame.

(Refer Slide Time: 55:27)

Swap Space

- Consider a system with 500 MB of main memory.
  - Can we run a program that requires 1 GB of main memory ?
  - YES
- Add an additional entry in the page table.
  - bit → Is the frame found in main memory, or somewhere else (???)
  - Hard disk (studied later) contains a dedicated area to save frames that do not fit in main memory. This area is known as the swap space.

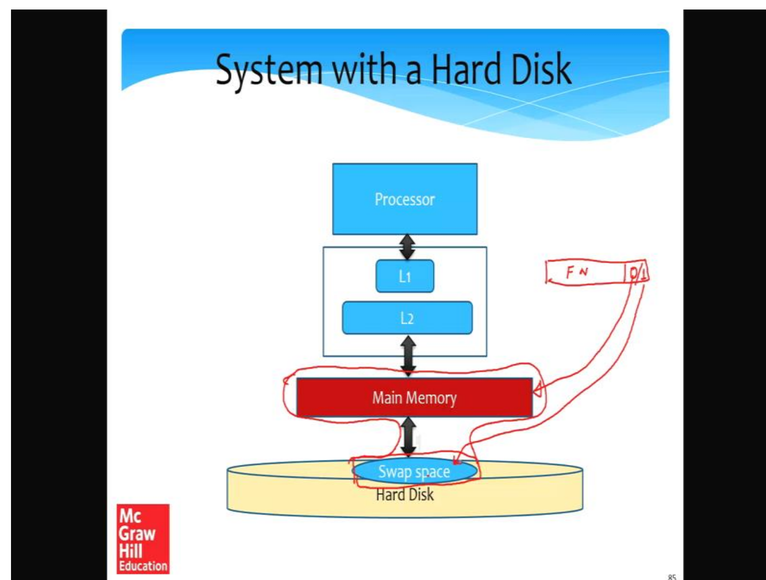
So, now let us consider a new term called swap space. So, let us consider a system with 500 megabytes of main memory. So, the question that we ask is can we run a program that requires 1 gigabyte of main memory. So, let us look at it you know 1 gigabyte of should not be main memory, but should be memory actually. So, let us assume that we have system, where we have a processor we have an l, we have an 2, and then we have our main memory, and the caches are inclusive where the size of the main memory is 500 megabytes, where our program is huge. So, it actually accesses in a 1000 megabytes. So, 1 gigabyte of main memory 1 gigabyte of memory is what it requires.

Can we run our program on such a system well? So, the answer as it turns out is yes, because if we do not allow, this will be really constraining our program, and we will also need to invest in the lot of memory. So, it is a better idea to have some way of doing this. So, what we can do is that we can add an additional entry in the page table. So, this will be the additional bit, it will say is the frame found is this particular frame found in main memory or somewhere else; so the page table. So, what does the page table entry look like or a TLB entry?

Given the page number it will contain the frame number, but what we are suggesting is that we can have a little bit more, or we can have some more data as well. So, we can have some amount of state they sells; that is this frame available in main memory, sorry m or is it available somewhere else or is it available somewhere else. So, in this case we can add more in a memory like storage devices to our system, and when the translating from virtual page to physical frame we can specify in exactly which storage device will you find this physical frame. So, one of this possible storage devices is main memory that we can have other as well.

So, let us consider the hard disk. So, almost all computers have a hard disk. So, we will study the hard disk later in chapter twelve, but for beginners it is a magnetic disk that stores data if the main memory is something of the tune of 8 gigabytes or 16 gigabytes, the hard disk is much more it is typically 500 gigabytes or 1 terabyte. So, it is much larger, but it does not use semiconductors it uses magnetic storage. So, it is much slower much slower. So, we can say that the hard disk can contain a dedicated area inside it, called the swap space right. So, in the swap space we can save from frames which are fitting in main memory. So, they can be saved in the swap space.

(Refer Slide Time: 58:48)

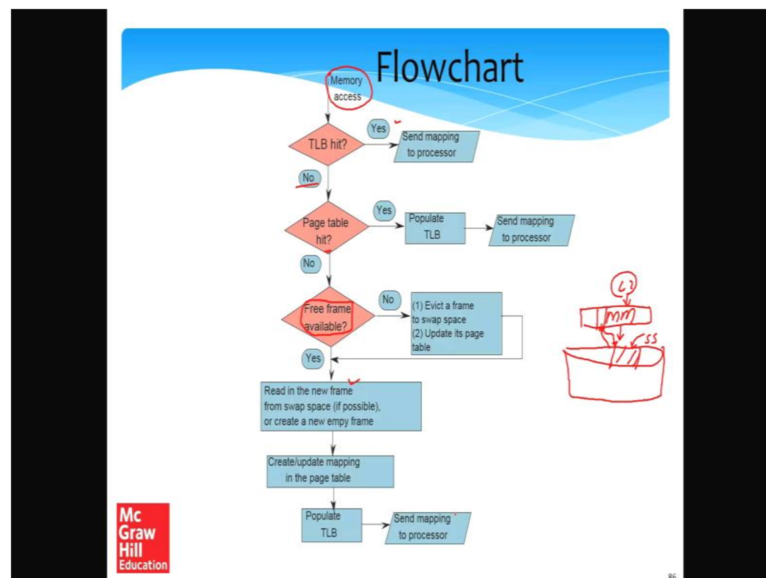


So, a conceptual diagram of this would be, that we have the processor then we have the l cache, we have the 2 cache below. The 2 cache we have the main memory this we are saying is somewhat constraint. So, if we make some additional space we can go to the

hard disk, which is the biggest storage structure that we have on machines, and a part of it, we can dedicate to actually storing some of the frames that are not fitting in main memory.

So, in this case the physical memory right then physical storage area is becomes a combinations of the main memory and the swap space right; so a combinations of this and that clearly. The main memory part is much faster, but we can still get little bit of additional storage in the swap space. So, what each space table entry will contain is that it will contain the frame number, and will contain a bit right. So, the bit defeat is 0. It can say that the frame is there in main memory, if it is one, it can indicate that the frame is there in swap space.

(Refer Slide Time: 59:58)



So, let us now look at s flow chart right of typical memory access. So, if we have a memory access can be a loaded store does not matter. So, if the TLB hit well and good that is the best case yes. So, we send the mapping to the processor we construct the physical address, and we send it to the memory system. So, if there is no TLB here we check the page tables, if there is hit in the page table, then they populate the TLB with a new mappings, and we also send the mapping to the processor; such that it can create a physical address. Now if there is no page table head, you know if this mapping is not there in the page table. So, we do not find an entry in the page table if this means that there is no mapping. So, the mapping means to be greater.

So, what we do, is that we check if a free frame is available in main memory right, in the in our frame list. Well if is available, yes and good we go to this stage; otherwise it is not available we go to main memory, and we evict a frame to swap space right. So, the assuming that we have main memory and. So, the 2 cache is connected to the main memory, and the main memory is connected to the hard disk which is huge. So, part of this is, let us say the swap space let me call it access.

So, we take one frame in main memory, and we essentially kick it out and put it in the swap space, we also update the page table for that frame saying that, it is in the swap space, and then we read in the new frame. Say if let us frame as some data in the swap space, what we can do is, that we can either create an empty frame or if we know what it is content should be, we read in that from the hard disk, and then we create an update the mapping in the page table. We populate the TLB, and we send the mappings to the processor, and the processor can use this mapping to create the new (Refer Time: 62:06) create a physical address and send it to the memory system.

(Refer Slide Time: 62:10)

Advanced Features

* Shared Memory → Sometimes it is necessary for two processes to share data. We can map two pages in each virtual address space to the same physical frame.
* Protection → The pages in the text section are marked as read-only. The program thus cannot be modified.

Now, let us take a look at some of the advance features that are possible. So, we know with art paging and virtual memory setup. So, sometimes it is necessary for two processes to actually share data, and we mind you this is purposeful this is deliberate that sometimes it might be necessary for process one, to some, send some data to process two. So, when will this be necessary well we can think of producer consumer situations

that we have? Let us say one process which is accepting request from the swap. So, this is p 1. It was accepting request from the swap, then we have a process p 2, which will get the request from process 1, p 2 will access some data base give some data to process p 1 and p 1 will send the data back to the client over the net, over the web, over the network link.

So, in this case p 1 and p 2 need to share some data, and this can easily done with our mechanism, what we can do is, for the same physical frame, we can map it to two separate pages; one pages is in the address space of process p 1, and one page is in the address space of process p 2. So, if process p 2 writes to this page, which is actually writing to this physical frame. Similarly process one write to this page which is writing to the same physical frame. So, in this way kind of indirectly, they can share data. So, if p 2 write something over here is in the same memory locations that p 1 can access, p 1 can read the values.

So, implementing shared memory in this way is very common and it is you know extremely common to use virtual memory for you know creating mechanism. So, processes to share data another thing are that, let us assume that we want to protect some regions of the program, and we do not want them to be modified. So, once the section that can be protected is the text section, which is where the instructions of the program are their right, and we do not want the instructions to be modified, because if somebody modifies the instructions, is like inserting a virus. So let say- if I modify these instructions, and I make them send random emails to random people on your net; that is going to create problems.

So, we want the program to be on modifiable. So, how do we do this? Well, the simple ways that we extend each page table entry or TLB entry with the frame number can have some additional data, and we also add a read only bit, if the read only bit is set to one, means that you know no other process can actually write to this memory region, you know just in case we are able to break into a program, it ensures that even the same process cannot write into this region.

So, essentially the text of the program the instructions of the program cannot be modified. So, this is one way of ensuring protection, and security in a virtual memory system that we can mark large parts of the memory space, as read only which means that,

once it is you know, it can only be read for the first time, it can only be written the first time, it is initialized after that no process can write to it, and this ensures that our programs remain the way they are, and they are not maliciously modified.

So, this brings us to the end of the memory systems chapter. So, we are done with memory systems. So, we are already to take on chapter number eleven, which is the chapter on parallel programming and multiprocessor systems.