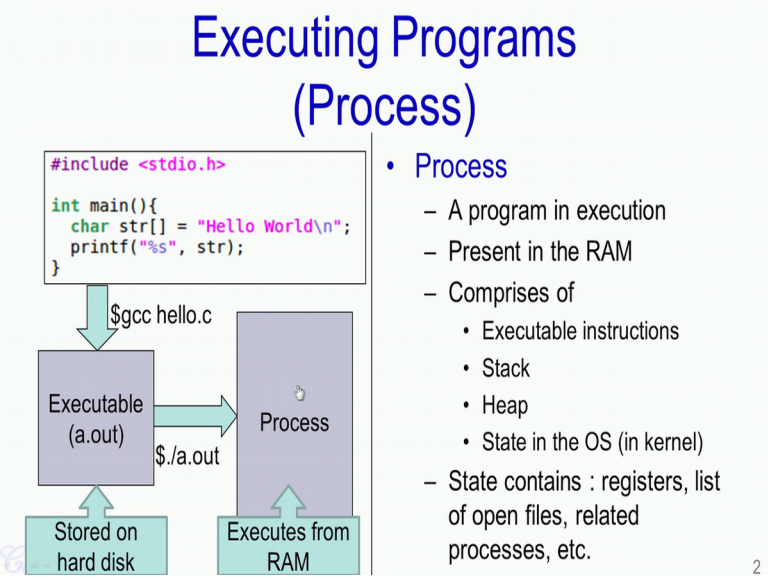
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 02
Lecture - 05
Memory Management

Hello. In a previous video (Lecture 4 video) we have seen, how the operating system has to manage the CPU because it is probably the most important resource in the system. Arguably, the second most important resource, rather the second most important hardware resource in the system is the Memory.

In this video, we will look at how the operating system manages the memory? Essentially, we will see how the operating system manages the RAM present in the system.

(Refer Slide Time: 00:56)

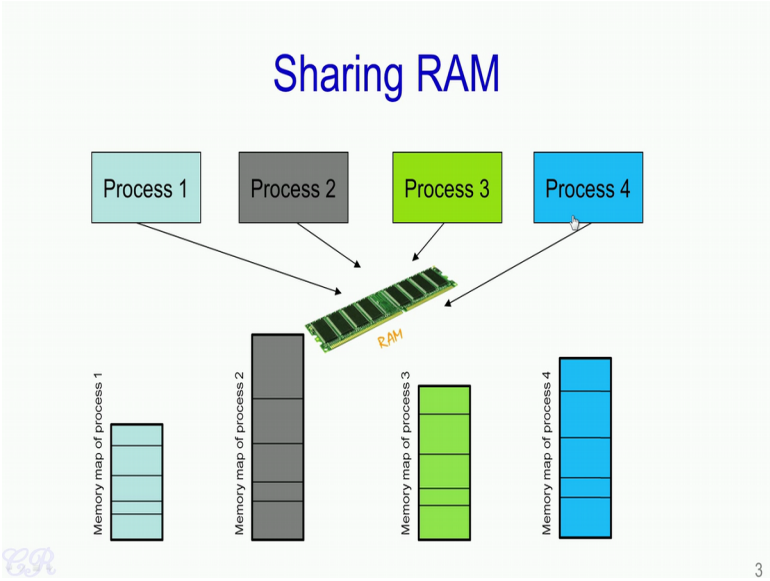


So let us review this particular slide (mentioned above) again. We have seen that when we take up a particular program and this could be any program written in any language and when we compile it, we will get what is known as an Executable.

Now, the executable will be stored on the hard disk and whenever a user runs this program or executes this program (mentioned in above slide), it creates what is known as a Process. So, this process is created by the operating system and it executes in RAM. So, essentially what the operating system would do is that the executable stored in the hard disk, would be loaded into the RAM and then on execution is passed to this particular program and this program will then execute in the CPU.

So, as we have seen this particular process (mentioned in above slide) which is present in memory comprises of several segments, such as the text segment which contains the Executable instructions, the Stack, Heap and also as we have seen some hidden metadata in the operating system such as registers, list of open files and related processes. So, all these actually constitute the process. Now, what is important for us is this process (mentioned in above slide) and it is presents in the RAM.

(Refer Slide Time: 02:45)



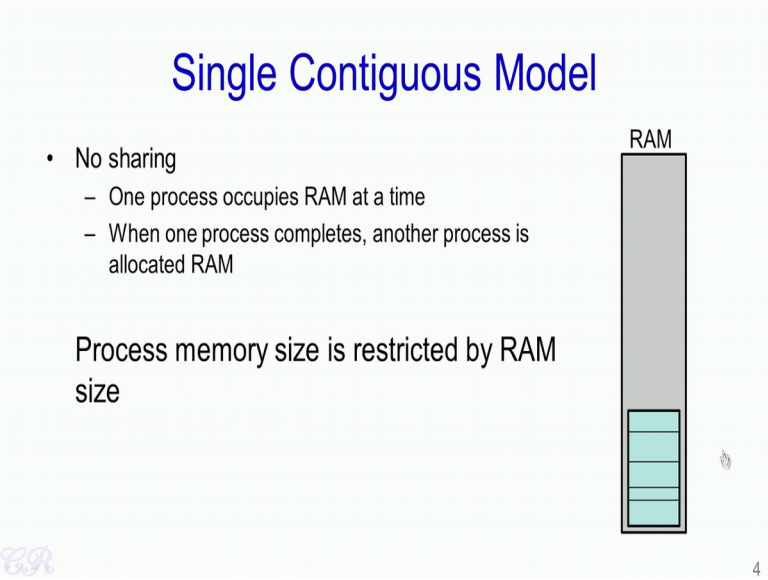
So as we know the RAM or Random Access Memory also called as the Main Memory is a limited resource in the system. So, each system would have something like 4, 8, 16 or 32 GB of RAM. At the same time we may have multiple processes executing almost simultaneously. So, like we have seen in the previous video (Lecture 4 video), these

processes may execute in a time sliced manner and if there are multiple processors present in the system, then these processors could also execute in parallel.

However, in all these cases, these processes and their corresponding memory map should be present in the RAM. Essentially the memory map corresponding to process 1, the memory map corresponding to process 2, memory map of process 3 and the memory map of process 4 should be present in the RAM, in order that these 4 processes execute in parallel or in a multi tasked environment.

Now, what we are going to see is how the operating system manages this resource (mentioned in above slide) or in other words the RAM such that it would allow multiple processes to execute almost simultaneously in the system.

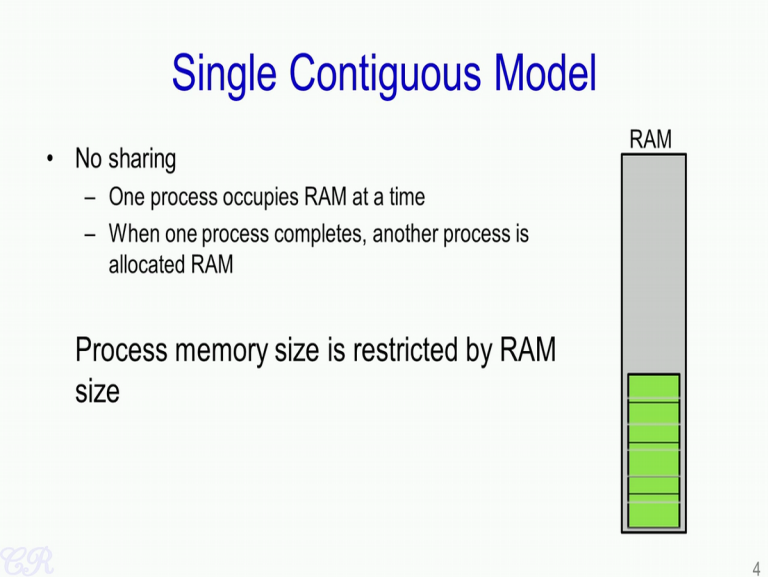
(Refer Slide Time: 04:25)



So, one of the most primitive ways of managing memory especially done for the older on a operating systems is what is known as the Single Contiguous Model. So, essentially in this we have a RAM over here (mentioned in above slide) which is the RAM of the system and what is ensure or what is done by the operating system is that this RAM is occupied by one process at a time. Essentially at any particular instant there is only one

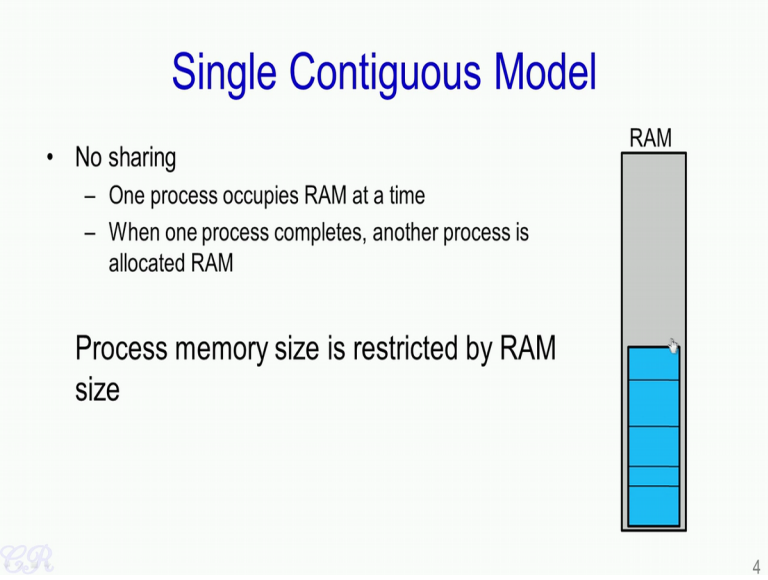
process and it is memory map that is present in the RAM. So, after this process completes executing will the next process will be loaded into the RAM.

(Refer Slide Time: 05:21)



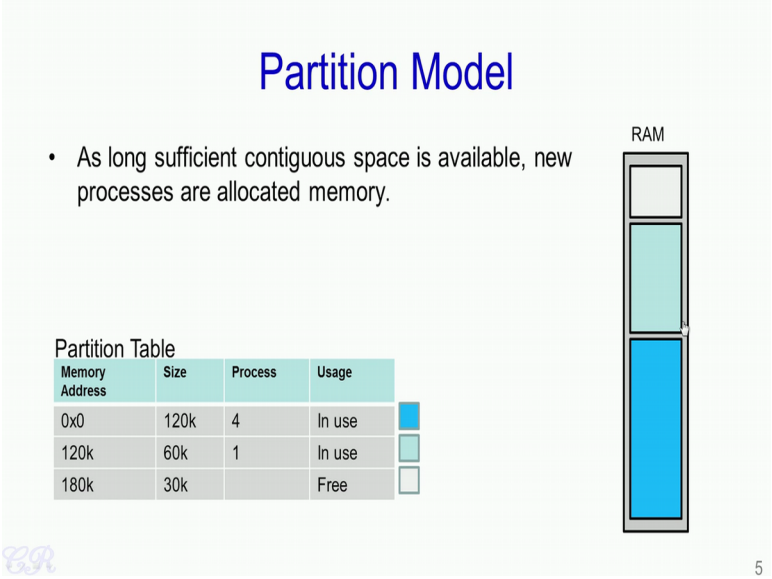
So, the drawbacks are quiet obvious.

(Refer Slide Time: 05:25)



The first is that we are forced to have a very sequential execution. When one process completes, only then the second process could occupy the RAM and so on. Another limitation of this particular model that is a single contiguous model is that the size of the process is limited by the RAM size. For instance, let us say we have a RAM which is of say 12 KB while this is seems to be a very small amount of RAM, this size of RAM is quiet common in embedded system. So, given this RAM of say 12 KB and let us say our process size is of 100 KB then it is quite obvious that this process cannot execute using this RAM. Essentially the RAM is not sufficient to hold the entire process.

(Refer Slide Time: 06:35)



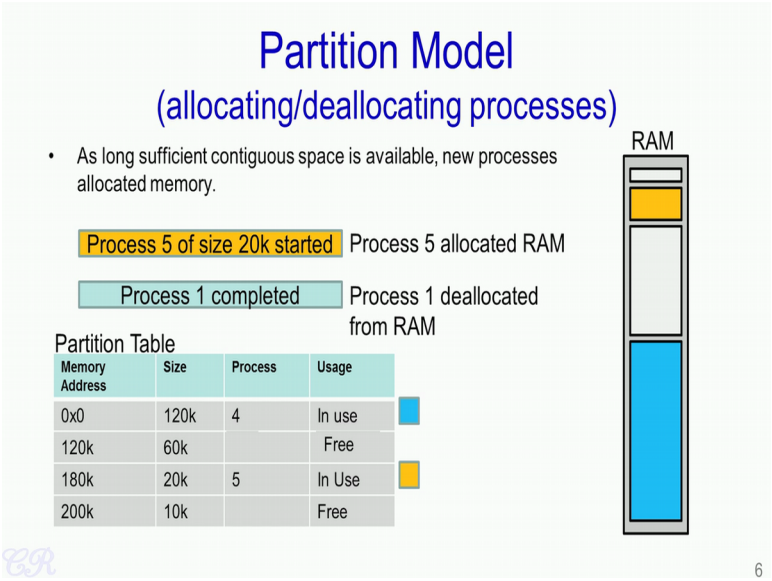
The next model that we will see, which is a slight improvement over the single contiguous model is what is known as a Partition Model. Essentially in this model at any instant of time, we could have multiple processes that occupy the RAM simultaneously. For instance, in this particular case we have two processes. This blue process and the green process (mentioned in above slide) that occupy the RAM and therefore, the processor could then execute this process as well as this process either in parallel or in a time sliced manner.

Now, in order to manage such a partitions, the operating system would require something known as a Partition Table. So, typically this partition table (mentioned in slide above)

would also be present in the RAM. It will be present in an area which is not shown over here. So, the partition table would have the base address of a process, the size of the process and a process identifier. For instance, the blue process has a memory or a base address of 0x0 indicating that is starting from 0^{th} location in the RAM and this process has a size of 120 KB. So, 120 KB means up to this point.

There is also a flag known as the usage flag which mentions whether this particular area in RAM is in use or it is free. For instance, let us take process 1 that is this one, the green one over here shown over here (mentioned in above slide). So, this process 1 starts at the memory location 120K that is this point and has a size of 60K. So, it extends up to 180K and this area is also in use. Now, there is another entry in the partition table which is specified as free. So, this starts at 180K and extends for a size of 30K. So, this corresponds to this white area over here (mentioned in above slide). So, the operating system could possibly use this free memory to run perhaps the third process and therefore, would be able to have three processes present in the RAM at the same time.

(Refer Slide Time: 09:34)

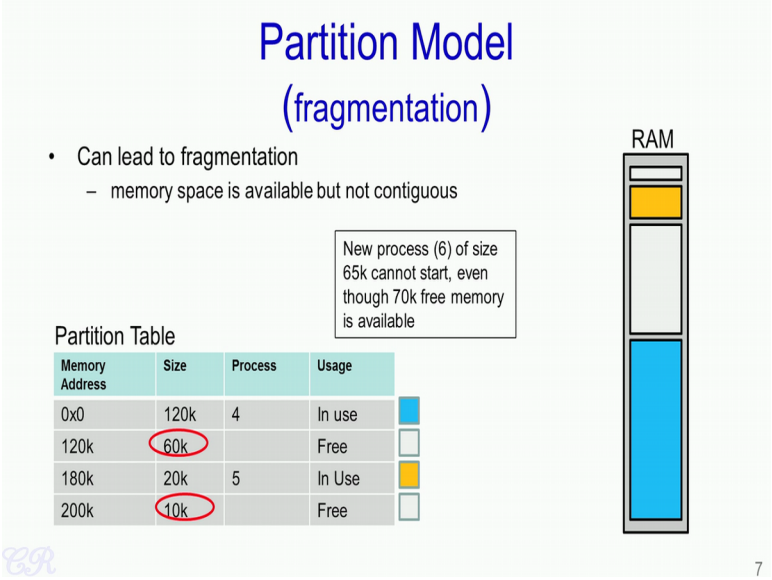


So, let us say a new process with the identifier 5 has just being created. So, the operating system would look into the partition table and create a particular entry for this new process. Now, since this process requires a size of 20 KB of RAM therefore, the

operating system will allocate 20 KB of RAM for that process. So, this allocation is done from the free space and as we see over here (mentioned in above slide) the 3rd process now gets end present in the RAM while the free space reduces in size. So, we now have the 10 KB of free space.

Similarly, when a process completes execution, for instance when process one completes its execution, the area in RAM that it holds will be de-allocated. Consequently the entry corresponding to the partition table will be free. So, this would lead to a free memory of 60K in the RAM. So, this corresponds to this particular area in the RAM (mentioned in above slide) which was used by process 1. While this technique of partitioning the RAM and therefore, allowing multiple processes to be allocated in the RAM is quiet easy to do. It could lead what to something known as Fragmentation.

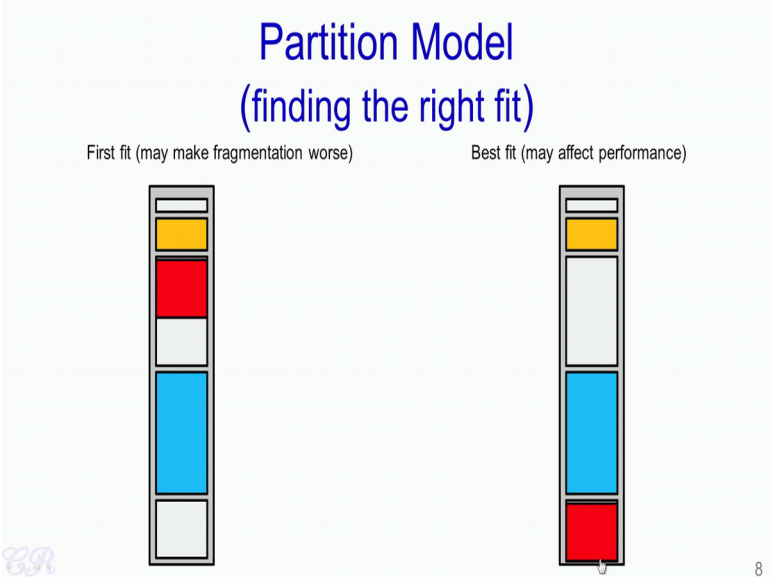
(Refer Slide Time: 11:18)



So, let us say this is how our RAM looks and as we seen we have a total of 70 KB of RAM which is free i.e we have 60 KB + another 10. So, that is 70 KB of free RAM memory. However, in spite of having 70 KB of RAM, a new process such as a new process with Id 6 which has a size 65 k cannot start even though 70 KB of free RAM is available and the reason for this is that the 70 KB of free memory is not in contiguous locations.

For instance, we have 60 KB of free memory present here and 10 KB of free memory present here (mentioned in above slide). So, individually each of these blocks of free memory is not sufficient to start a new process of 65 KB. So, this is what is termed as fragmentation and could result in what is known as the under utilization of the RAM.

(Refer Slide Time: 12:37)



Now, let us assume that when a new process arrives, there is sufficient amount of free spaces that are present in the RAM memory. So, next the operating system has to decide that which among this free memory should the new process be loaded into. For instance over here, there is a new process which has just arrived and we have 3 blocks of free memory, this one, this one and this one (mentioned in above slide). So, which of these three blocks of free memory in the RAM should the new process be loaded into?

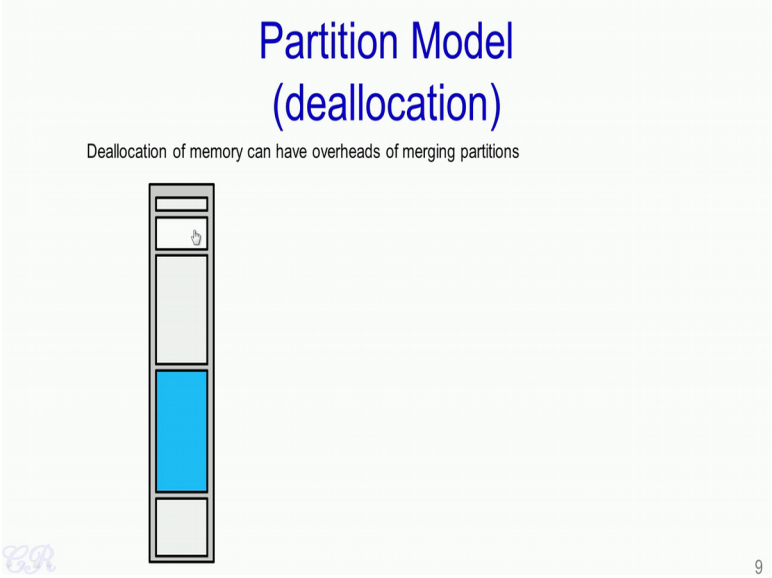
There could be several algorithms to achieve this. So, one of the most simplest algorithm is something known as the First Fit. So, with this First fit algorithm what the operating system would do is scan the free blocks in RAM starting from the top and use the first available free block which is at least as large as the processes requirements. So for instance over here, this RAM is scan from the top and it was see that the first free block is too small for the process to fit in. Therefore, it is not allocated here.

The next free block is large enough to allocate this process. Therefore, the process is allocated here (mentioned in above slide). While this First fit allocation algorithm is extremely easy to perform from an OS perspective, it could make fragmentation worse. Essentially with the First fit, what we are doing is that we are breaking the amount of free blocks into smaller and smaller chunks. Thus individually each chunk will not be able to get a single process, thus making the fragmentation worse.

The other algorithm that we will see is known as the Best fit algorithm and this performs considerably well with respect to the fragmentation. Essentially it will not fragment the memory as much as the first fit algorithm. So, what happens here is that when a new process enters or is created, the operating system will scan through all free blocks that are available and choose the block which is the best fit for that process. So, in this particular case as we have seen earlier as well, this particular free block is too small (mentioned in above slide) to cater to this new process. This free block is too large while the 3rd free block is just correct.

So, the operating system would allocate this process into this free block as shown over here (mentioned in above slide). While the best fit algorithm efficiently utilizes the available RAM reducing the fragmentation issue, there may be a performance hit. Essentially, it may result in a deterioration of performance. The reason being is that now the operating system has to scan through every free block that is available and needs to make a decision about which free block is best for the new process and this would take some time.

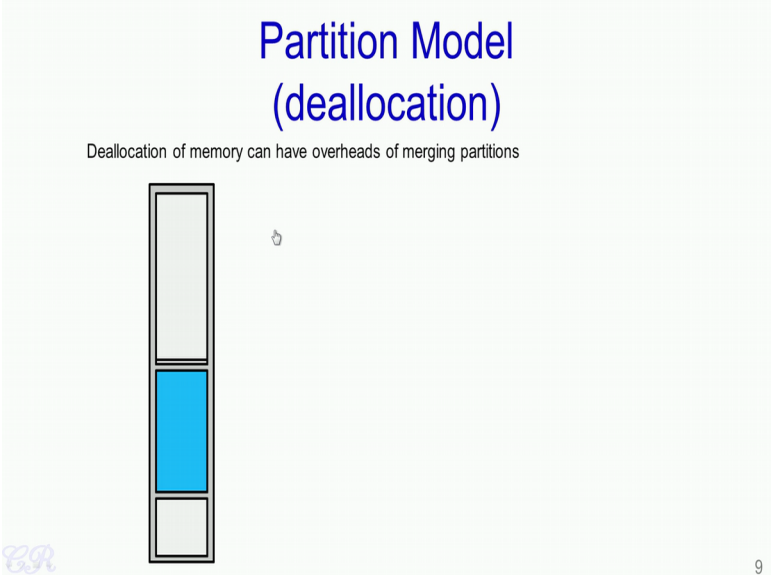
(Refer Slide Time: 16:28)



Another issue with the partitioning model is the case of deallocation. So, essentially deallocation would occur when a process completes its execution or is terminated and has to be removed from the RAM in order to allow a new process or another process to execute from the RAM.

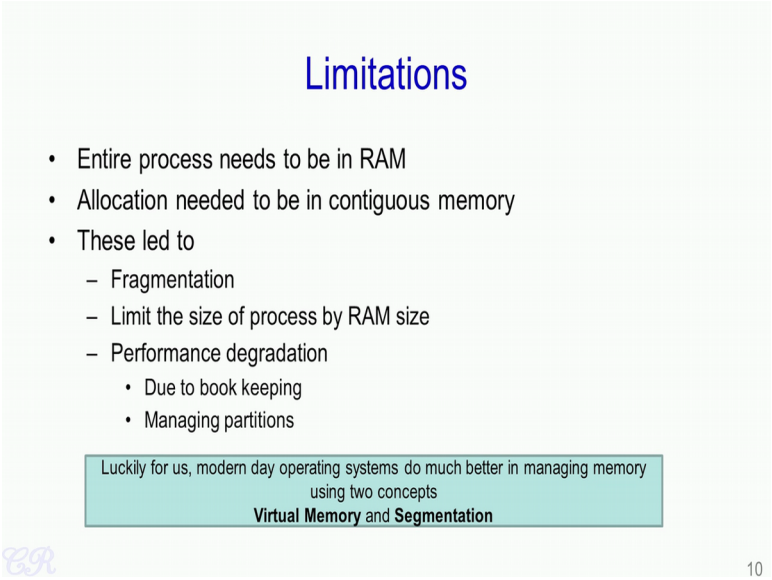
So, deallocation would require that a new free block to be created and it would result in the free flag set corresponding to this block in the partition table. So, what the operating system now has to do is that it should detect that there are indeed three contiguous free blocks that are present in the RAM and as a result of this, it should detect these three contiguous free blocks (mentioned in above slide) and be able to merge these three blocks in to one single block.

(Refer Slide Time: 17:34)



The advantage of merging it into one thick free block is that now this larger free block could cater to a larger process and thereby reducing the effect of fragmentation.

(Refer Slide Time: 17:51)



So, thus we have seen that the major limitation of the single contiguous model as well as the partition model is that the entire memory map of the process needed to be present in

RAM during its entire execution. So, all allocation for the entire process needed to be in contiguous memory and because of these issues, it had led to fragmentation, limit on the size of the process due to base on depending on the RAM size and also performance degradation due to book keeping and also the management of partitions. So, luckily for us modern day operating systems do much better in managing memory. So, most modern day operating systems use two concepts that is virtual memory and segmentation.

So in the next few videos, we will look into these memory concepts rather these memory management concepts and we will also see how the Intel x86 processors manage memory.

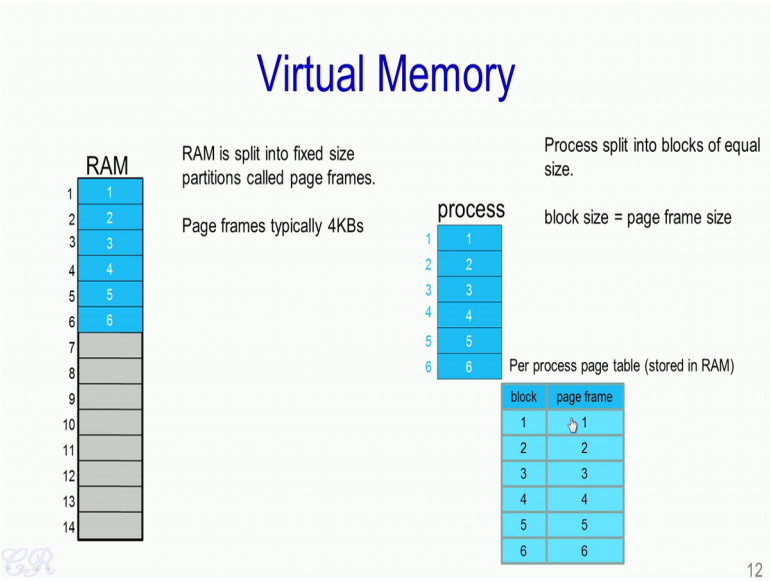
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 02
Lecture – 06
Virtual Memory

Hello. In this video, we will look at Virtual Memory; which is by far the most commonly used memory management technique in systems these days.

(Refer Slide Time: 00:31)



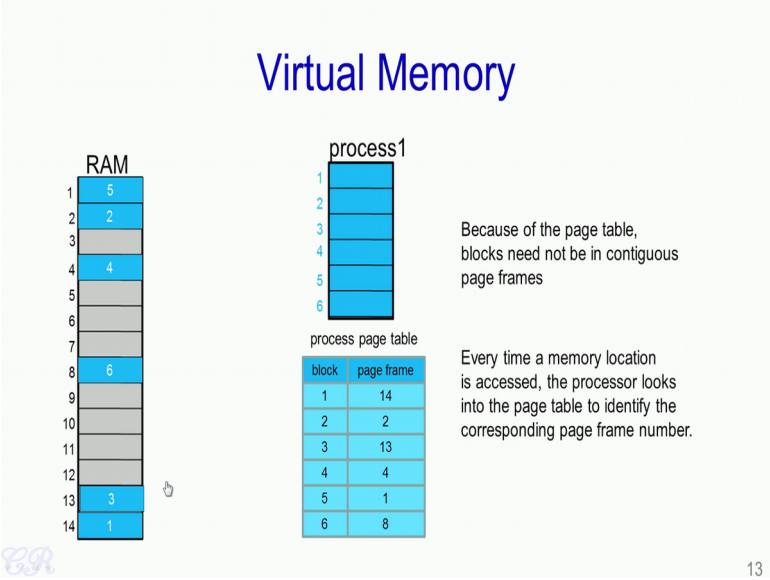
So in a virtual memory system, the entire RAM which is present in the system is split into equal sized partitions called Page frames. So typically, page frames would be of 4 kilobytes each. So in this RAM for instance, we have 14 page frames; which are numbered 1 to 14. And, each of these page frames have the same size.

In the previous Intel processors, all pages were fixed at 4KB. But, in more recent processors we could have pages or page frames, which are of a larger size. In a similar way, the process which executes in the CPU or the process map corresponding to the process is also split into equal sized blocks. Now, the split of a process is in such a way that the block size of a process that is this size (mentioned in above slide) is equal to the page frame size; that is the size of each block in this process is equal to the page frame size.

Now because we have split the RAM like this (mentioned in above slide), as well as the processor's memory in a similar way, what we can then do is allocate blocks in a process to page frames in the RAM. Additionally, what the operating system maintains is a table. This table is also present in the RAM and not shown over here. But, what it contains is the mapping from the process blocks to the page frames. For instance over here, the mapping is very simple; block 1 of the process gets mapped into page frame number 1, that is, block 1 of the process gets mapped into page frame number 1, block 2 gets mapped to page frame 2, block 3 gets mapped to page frame 3, and so on.

Now this (mentioned in above slide), I would like to highlight again; is a per process page table. So per process means that every process running on the system will have a similar page table as shown over here.

(Refer Slide Time: 03:23)



Now because we have such a table which provides the mapping between blocks of a process to the corresponding page frame, what we can then do is we could have any kind of mapping that we choose. For instance, now we have block 1 of the process present in block 14 that is over here (mentioned in above slide) block 2 of the process present in page frame 2, block 3 of the process in page frame 13 and so on.

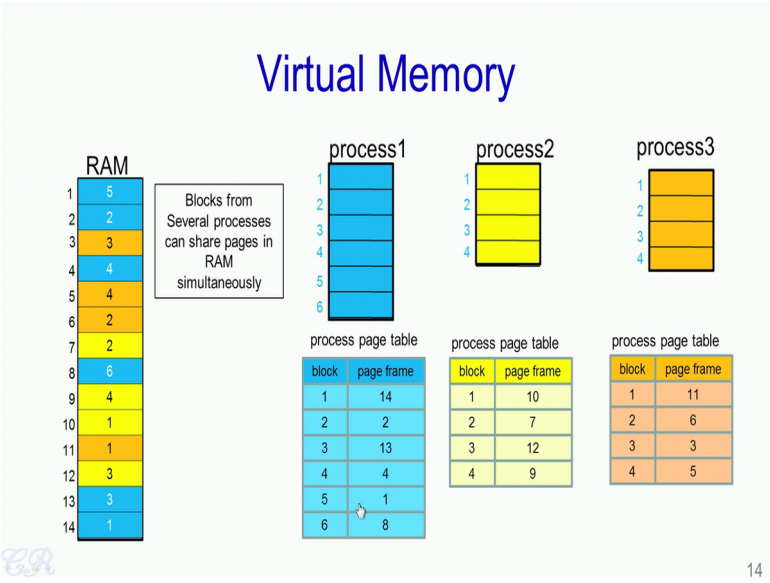
Essentially, what we are able to achieve with this particular process page table is that the blocks of the process need not be in contiguous page frames in the RAM. However, the overhead that we incur is that every time a particular memory location needs to be

accessed in the processors memory map. The CPU would need to look up the processors page table, obtain the corresponding page frame number for that particular address and only then can the RAM be addressed or accessed. For instance, let us say that the program executing in the CPU accesses a memory location inside a block 3 of the process. So, this could be an instruction or a load or store to some data.

Now, when such an operation is executed a memory management unit present in the processor would intercept this access. And then, it is going to look up in to the page table and find that the corresponding page frame, which towards block number 3 is 13. Then, it is going to generate something known as a physical address, which will then look into the 13th page frame in RAM. As a result, every memory access has the additional overhead of looking into the page table, before the access into RAM can be made possible.

So this look up into the page table is the extra overhead. And, typically this overhead is partially mitigated by using something known as a TLB cache or a Translational Lookaside Buffer cache. So, we will not go into details about the TLB cache. But, for our understanding with respect to the course we are studying, we need to remember that every memory access or every load or store of instructions fetched by the process during its execution would need to look up a particular page table and only then can the RAM be accessed.

(Refer Slide Time: 06:29)



Now, what we mentioned was that every process executing in the system would have its own process page table (mentioned in above slide). Thus, if you have a second process that is process 2, a process 2 will also have associated page table; process 3 will also have its associated page table. Now, these processes or these processes as we have seen the memory associated with these processes are present in the user space region of the memory. However, the process page tables are present in the Kernel's space. And therefore, any program you write in the user space, but not be able to determine what the process page table mappings are.

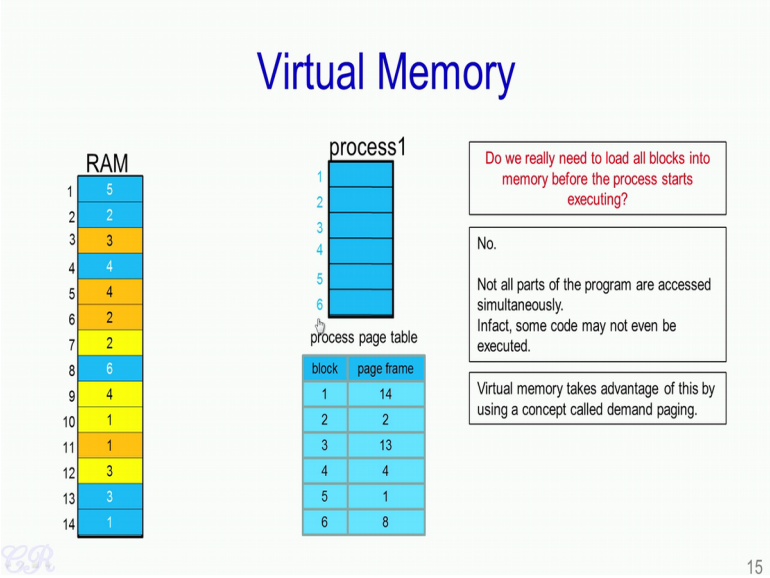
Now, since we have multiple processes executing in the system, each process could have various page frame mappings. For instance, (mentioned in above slide) process 1 has a mapping of all the blue page frames in this particular RAM; while process 2 has all the yellow page frames; when the third, process 3 has all the orange page frames.

Now, what we can see is that we are capable of sharing the RAM with multiple processes simultaneously. Now, assume that we have a system with a single CPU. This means that at a particular point in time, only a single process will execute. Now, this process will continue to execute until its time slice completes.

So, during the execution of the process, that is when for instance, process 1 is executing in the CPU, this particular process page table (mentioned in above slide) will be the active page table. Therefore, any instruction or data which is loaded or stored will have, will look up this particular process page table to get the corresponding page frame in RAM. When there is a context switch from process 1 to say process 2, it would be this (mentioned in above slide) process's page table, which will become active.

So, any address that is accessed in process 2 will get the corresponding page frame number from this (mentioned in above slide) active page frame. Similarly, when process 3 gets executed in the CPU, this (mentioned in above slide) process's page table will be the active page table. So, what we can see is that it is not possible for process 2 to access any of the page frames corresponding to process 1 or process 3.

(Refer Slide Time: 09:37)

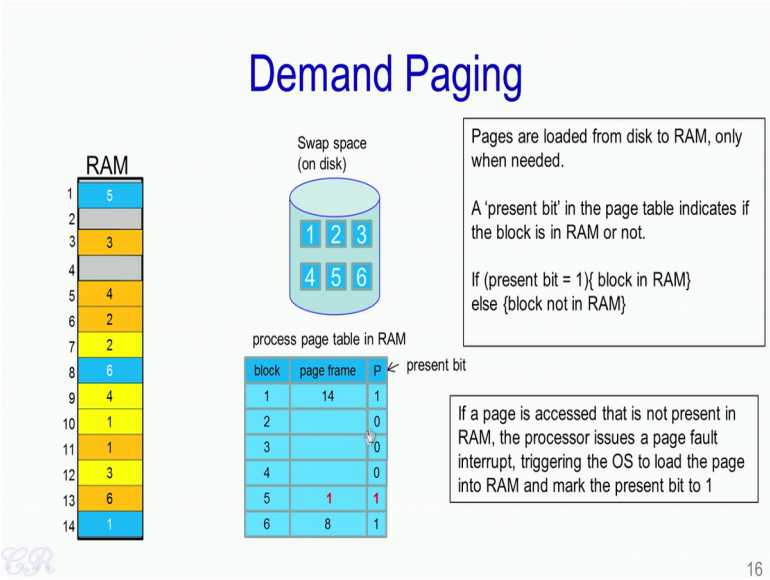


The next thing we would look at is that do we really need to load all blocks into memory before the process starts to execute? So, in the earlier video we had seen that when a process executes, the entire process was loaded into a RAM. So, now we ask ourself that is such a loading actually required? And, the answer is no. Essentially, what we notice about programs is that not all parts of the programs are access simultaneously.

In other words, there is a locality aspect in the way a particular program executes. What this means is that if some instructions are executed in let us say this particular block (mentioned in above slide), it is highly likely that the next set of instructions or the next few instructions that follow this would also be present in this block. So, once there is a instruction to another block, say block 3, it is quite likely that because of the locality of the program, there would be quite a few instructions executed from this block and so on.

In fact, you may have, and it is quite often the case that there may be several parts of the process memory, which are not even accessed at all. That is, they are neither executed, loaded or stored from memory. So, what the virtual memory concept does is that it takes advantage of this locality aspect during the execution by using a concept known as Demand Paging.

(Refer Slide Time: 11:43)



In a demand paging scheme, what would happen is that we would have on a secondary storage device like a hard disk, there would be a particular space allocated as the Swap Space. So, in this swap space (mentioned in above slide) all blocks of the executable will be present. So, for instance if the process has 6 blocks, 1 to 6 and all the 6 blocks will be present in the swap space. And only on demand, will blocks be loaded from the swap space into the RAM.

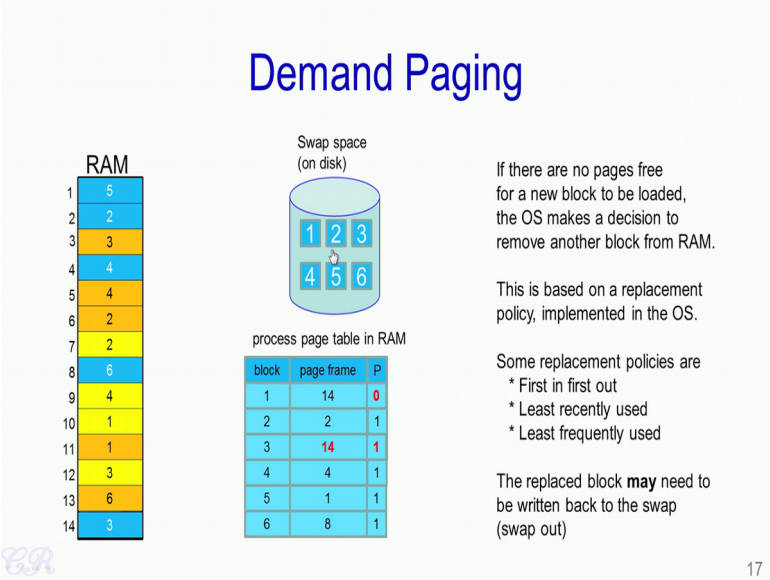
So, now let us assume that there is an instruction which is executing in page frame 8 that corresponds to block 6 of the process (mentioned in above slide). And, it has resulted in a load or a store to a particular memory location in block number 5. So, what happens is that the processor is going to look up the process page table corresponding to block number 5 and see that the present bit is set to 0. So, this would result in something known as a Page fault trap, also called as the Page fault interrupt.

Now, the page fault interrupt would then trigger the operating system to execute. Now, the operating system will then detect that this was a page fault interrupt. And, it will determine the cause of this page fault interrupt and it will load the corresponding page from the disk into RAM. Consequently, it will also modify the process's page table.

For instance, over here a value of 1 is added to the page frame number and the present bit is set to 1. Now, all later accesses to this particular block 5 will not cause a page fault because the present bit is set to 1. However, the first accesses to block 2, 3 and 4 would

result in page faults. And, it would cause the operating system to execute and load the corresponding blocks into the RAM. Consequently, the page tables for that particular process will be updated.

(Refer Slide Time: 14:10)



So now we have reached a state where every page frame in the RAM occupies some block. So, these blocks could either be from the process 1 or process 2 or process 3. Now, what happens if let us say there is a memory access to process 1's 3rd block (mentioned in above slide). So, as we can see that since the present bit is set to 0 in the process page table, it would result in a page fault interrupt that occurs. And, it would trigger the operating system to execute.

Now, the thing is what will the operating system do? So, in order to cater to this particular memory access in the 3rd block, the operating system would need to remove one of these pages. Essentially, it needs to clear up one of these page frames in order to make way for the new block. So, the obvious decision that the operating system would need to make is which of these blocks should it remove, in order to make way for the new block.

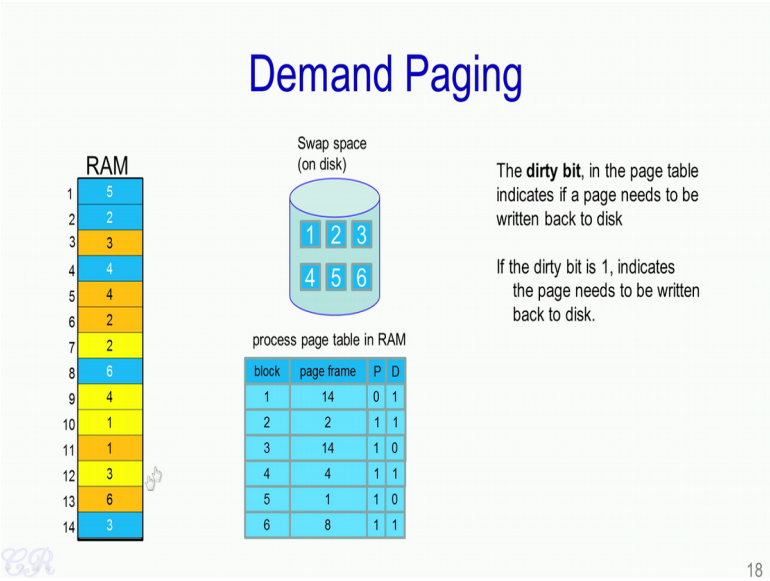
So, this choice is based on some algorithm present in the operating system. So, these are known as the Page Replacement Algorithm. And, there are quite a few choices for that; such as the First in first out, Least recently used or Least frequently used. So, based on the decision that the replacement algorithm makes, the OS would swap out a block from

the RAM and replace it with the third block that was recently accessed.

Consequently, the present bit corresponding to the block that has been removed will be set to 0. So, we had removed, for instance, the block number 1 from the RAM to the swap. And therefore, the present bit corresponding to block one will be set to 0 (mentioned in above slide). Now, block 3 which has just got loaded into the particular page frame will have its page frame number set and the present bit set to 1. So, this process of removing a block from the RAM and replacing it by another is known as the swap out and swap in, respectively.

The process of moving a block from the RAM to the disk is known as a swap out process; while, the process of loading a block from the disk to the RAM is known as swap in. Now, during the process of swap out, that is storing block 1 to the hard disk, all the changes that happened during the execution of the process (mentioned in above slide), all changes that occurred in block 1 will get updated in the swap space. Thus, the block which eventually gets stored in the swap space will have the latest view of the data. So, the next thing that we are going to ask is that if the swap out, that is, the copying of the block 1 from the RAM to the disk is actually required?

(Refer Slide Time: 18:12)



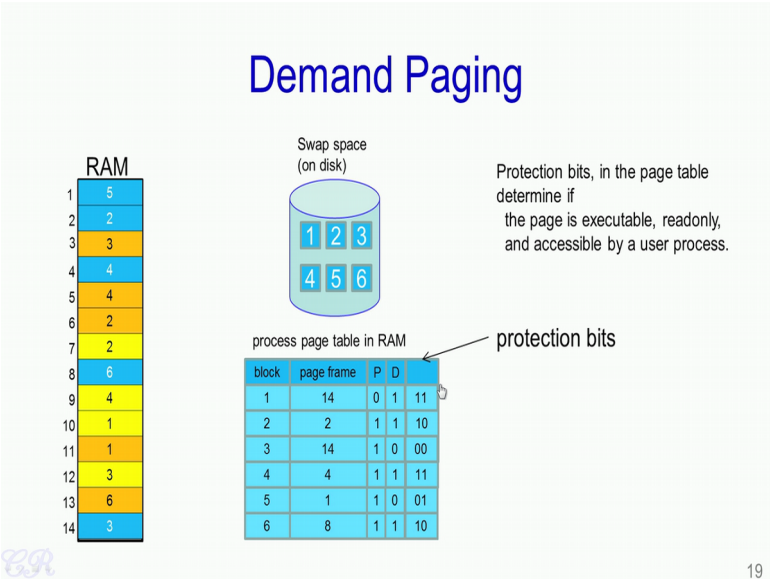
Let us assume that block 1 which was present over here (mentioned in above slide) in the 14th frame had no data that is changed. For instance, it could be corresponding to instructions or read only data. And therefore, none of this data actually changes. In such

a case, the swap out process of copying the contents of this page frame back to the disk will not be required. On the other hand, suppose we had data in block number 1 which was modified, so this would indicate that the copy of block 1 in the RAM is different from that in the disk. And therefore, the entire page frame should be swapped out and return back into the disk.

So, how will the operating system know whether a swap out is required or not? So, this is done by an additional bit, which is present in the page table known as the D-bit or the dirty bit. So, the dirty bit essentially indicates whether the contents of a block in the RAM has been modified with respect to its contents in the disk or in the swap space.

So, if the dirty bit is set to 1 (mentioned in above slide), it indicates that indeed there is a difference between the contents in the RAM corresponding to the contents in the swap space, and therefore during a page fault, the contents of the page frame will be returned back to RAM. If the dirty bit has a value of 0, then the entire swap out process will not be required. It will just require to swap in the new process and replace that corresponding block in that corresponding page frame.

(Refer Slide Time: 20:19)



In addition to the p and d bits, that is the present and dirty bits, the page table for a process has some additional bits known as the protection bits. So, these protection bits would determine various attributes for a particular block. So for instance, the process of the operating system could set various blocks as executable; which means that the data

present in that block corresponds to executable code.

And therefore, the CPU could then execute it. Other bits could also tell whether the block is read only and therefore, modifying of any data in that block will lead to a fault or an error. And, bits over here will also determine whether a particular block corresponds to an operating system code or it is a regular user code. So, these additional information is present by the protection bits.

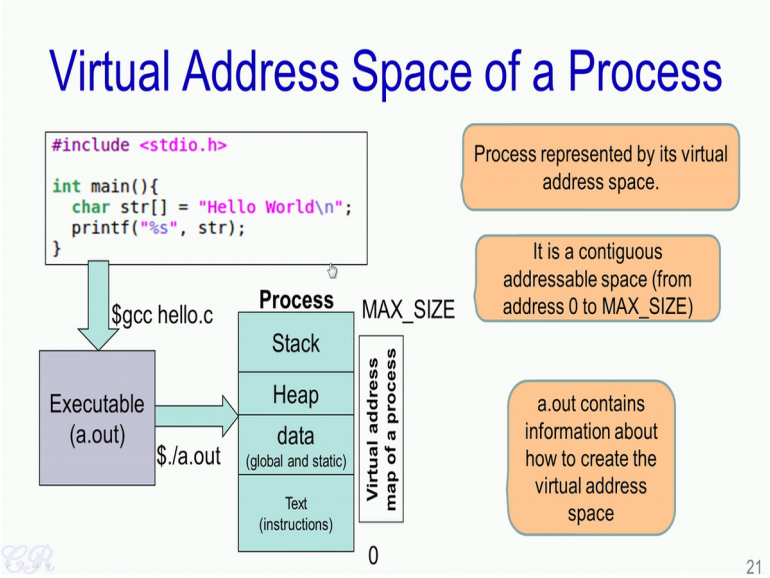
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 02
Lecture – 07
Virtual Memory (MMU and Mapping)

Hello. In this video, we will discuss how the virtual memory mapping takes place between the Virtual Memory and the RAM.

(Refer Slide Time: 00:32)



So essentially, let us start with this very famous slide by now; that when you write a program and compile it, you get the executable a.out. And, when you execute this particular executable, a.out executable, the operating system will create a process for you. Now, the process is represented by what is known as the virtual address space. Now, the virtual address space is a contiguous address space starting from 0 and extending up to an address, defined by something known as the MAX SIZE. This MAX SIZE comes from the xv6 nomenclature.

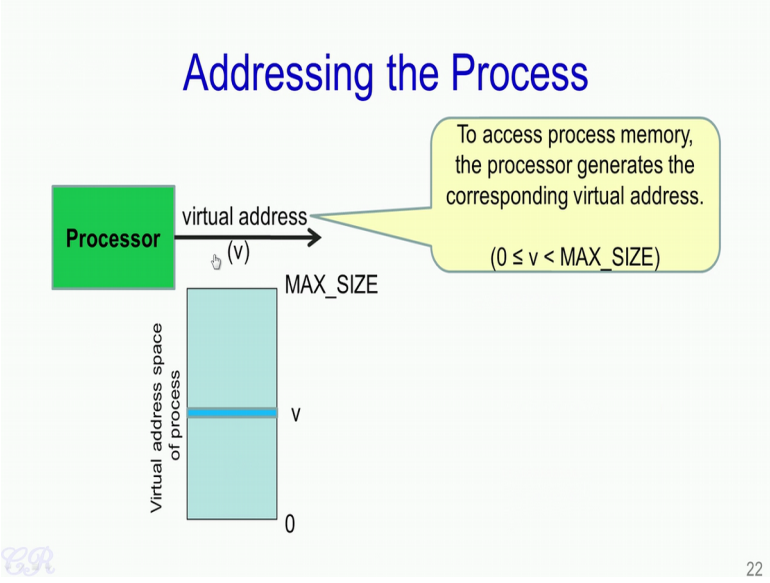
Now, within this contiguous virtual address space for the process are various sections like the Text, Data, Heap and the Stack. So, all these sections are created by the operating system. Essentially, the operating system will need to know where the heap would start, where the stack would start and so on. So, how does the operating system know all these

information? So, this information is given to the OS through the - a.out. Essentially what happens is that when the program is compiled and linked, the compiler will put a lot of these information in the executable a.out.

So when executed, the operating system will read out this information and then determines where in the virtual address space should these various segments be present. So, now when this process is executing all these instructions corresponding to 'printf' for instance, gets executed and all addresses corresponding to 'str' is accessed (mentioned in above slide). So, it may be noted that all these addresses, all these instructions, would be mapped into this particular virtual address space. For instance, if I were to print the address of "Hello World", essentially the address of this 'str' then the result would be the virtual address with respect to this particular mapping.

So, what we will next see is how this virtual address mapped; gets mapped into the main memory of the system. So, let us say that the processor wants to access this particular string 'str', at least the base address of this string, and as we know that would contain the letter 'H'.

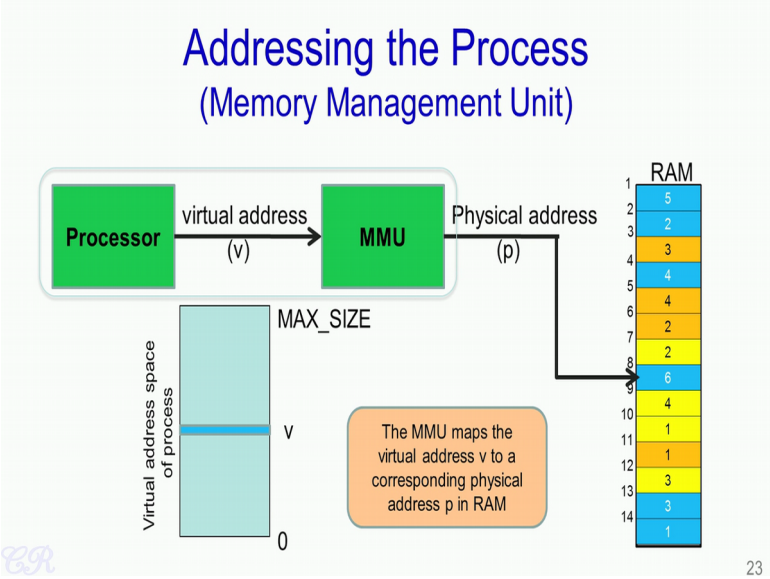
(Refer Slide Time: 03:41)



So, what would happen is that the processor will put out a virtual address on its bus. So, let us say this virtual address is 'v' and this virtual address could be in the range from 0 <= v < MAX SIZE. So, MAX SIZE is the largest address that a user space process could

have in its virtual address space. So said another way, the virtual address ‘v’ would be some address in the virtual address space of the process.

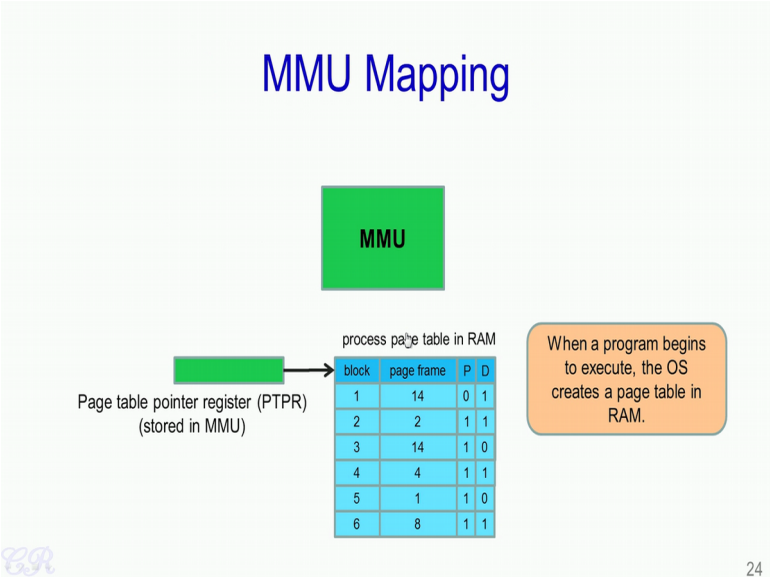
(Refer Slide Time: 04:19)



Next, another unit in the CPU known as the MMU or the Memory Management Unit, so this entire thing (mentioned in above slide) would typically be in the same package or jointly called the CPU or processor would then take the virtual address and convert it to a physical address ‘p’, which will then be used to access the RAM.

So, what we will see next is how this virtual address gets converted to the corresponding physical address. So, it may be noted that the virtual address corresponds to the virtual address map, but the physical address correspondence to one physical address in the main memory or the RAM.

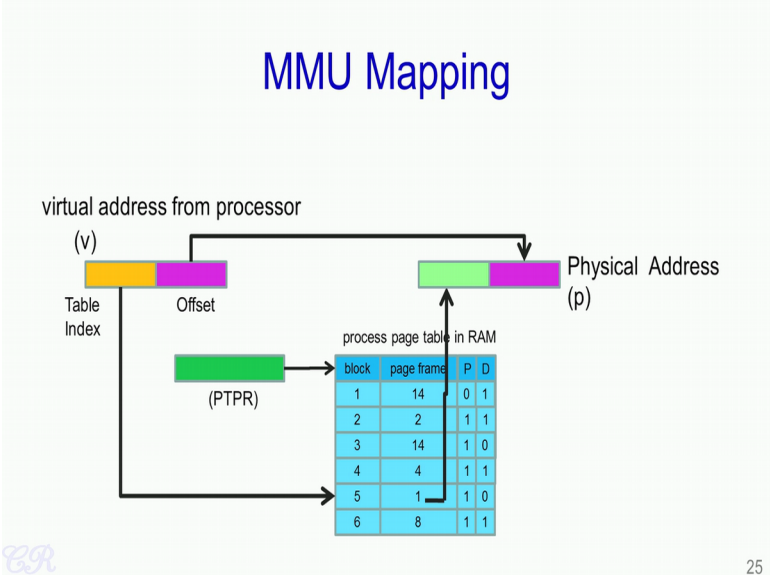
(Refer Slide Time: 05:09)



So when a process begins to execute, the operating system would create a page table for that process in RAM. So, this is the page table (mentioned in above slide). And as we have seen in the previous video, the page table holds the mapping from the virtual blocks or the virtual address space of the process to the physical page frames. And, we have other bits like the present bit, dirty bit and the protected bit which is not shown here.

Now in the memory management unit, there is a register known as the Page table pointer register or PTPR. So, in the Intel systems this PTPR or page table pointer register is known as the CR3 register. And, we will see more details of this in a future video. So, this PTPR register present in the MMU will have a pointer to the process page table. So let us see how these things are used to provide the mapping from the virtual address space to the physical space.

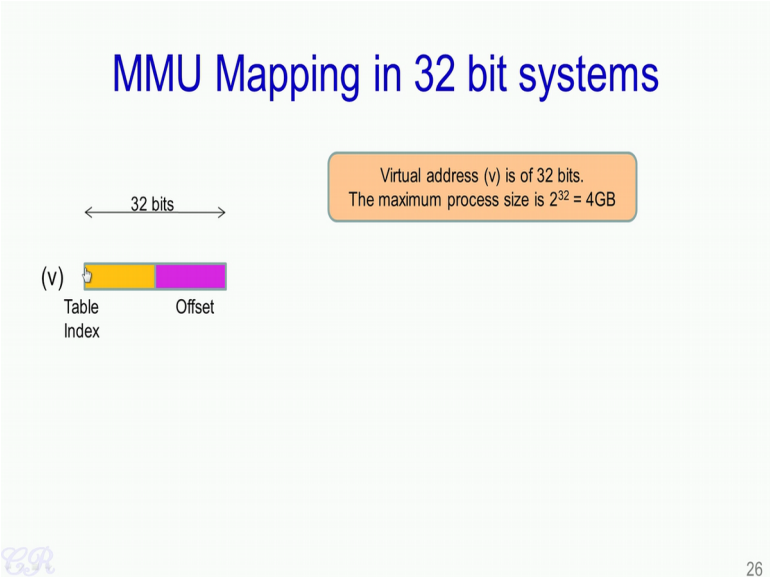
(Refer Slide Time: 06:18)



So, what happens is that the virtual address which is sent out by the processor core comprises of two parts. It has a table index, which are a few bits typically, they are more significant bits of the address and an offset. So, when the MMU obtains this virtual address, it is going to look up the process page table, corresponding to the offset present in the table index (mentioned in above slide). So, the PTPR would be the base address of this particular process page table and, the higher bits in the virtual address would correspond to the offset in the process page table. So from here, the corresponding page frame is taken and it forms part of the physical address.

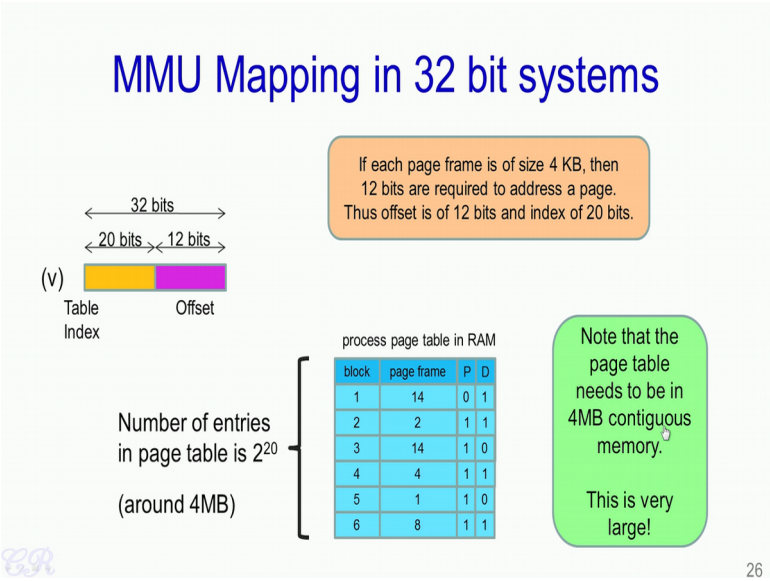
Now, the second part of the physical address is directly taken from the offset. So, in this way we would then create what is known as the Physical Address. And, it is this physical address that is used to access or to address the RAM on the main memory.

(Refer Slide Time: 07:40)



So let us see how this MMU mapping works for a 32-bit system. So in a 32-bit processor, the virtual address space could be at most 2^{32}, that is, 4GB. The virtual address is of 32 bits as shown over here (mentioned in above slide).

(Refer Slide Time: 08:03)



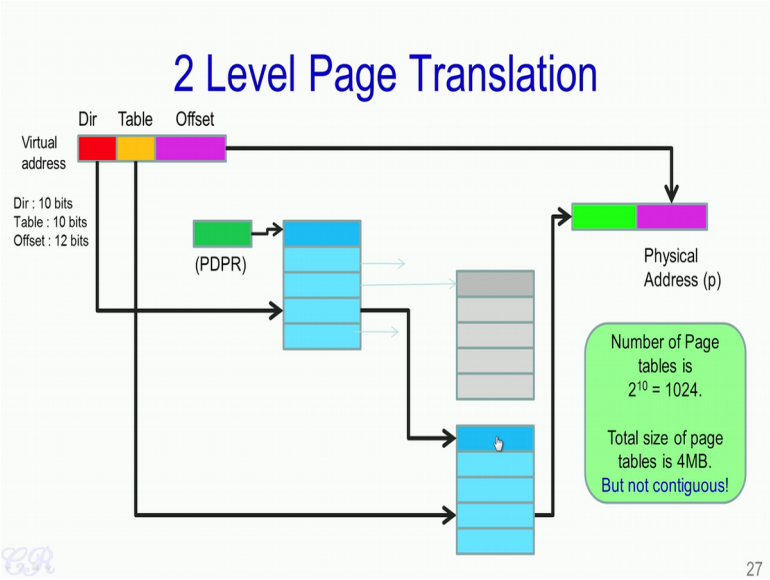
And, this is typically divided into two parts; the table index and the offset, which is of 20 bits and 12 bits respectively. So, how did we get this particular thing is by the fact that each page frame as we had mentioned in the earlier video is of size 4KB. Thus, to access any offset within the page frame would require 12 bits. 12 bits because 2^{12} is 4096, which

is 4KB and therefore, we have an offset which is of 12 bits, which essentially would give the offset within a page. The remaining bits that is, 32 - 12, which is a 20 bits, would be used for the table index. Thus, the process page table would be, which is indexed by these 20 bits would have 2^{20} entries.

Now, assuming that each entry is of 4 bytes, then the entire size of this page table is 4MB. And, what essentially is required is that this page table has to be contiguous, that is we need to have these 4 MB of process page table to be in contiguous memory locations.

So, why do we need it to be contiguous is essentially that the table index is added to the process page table pointer, in order to get the location of a particular block and the corresponding page frame. And therefore, it needs to be in contiguous memory. Thus, we see that each process that executes on the system will have the additional overhead of having a 4MB process page table, while 4 MB is not a very large space with today's RAM sizes of 32 and 64GB. However, the requirement that it needs to be contiguous could, would be an issue.

(Refer Slide Time: 10:36)



So, what some systems do, especially the Intel systems is to have 2 level page translation. Essentially, instead of having just a table and offset for the virtual address, now we have 3 components in the virtual address. We have a directory entry which is of 10 bits, a table entry which is also of 10 bits and the offset which as usual is of 12 bits.

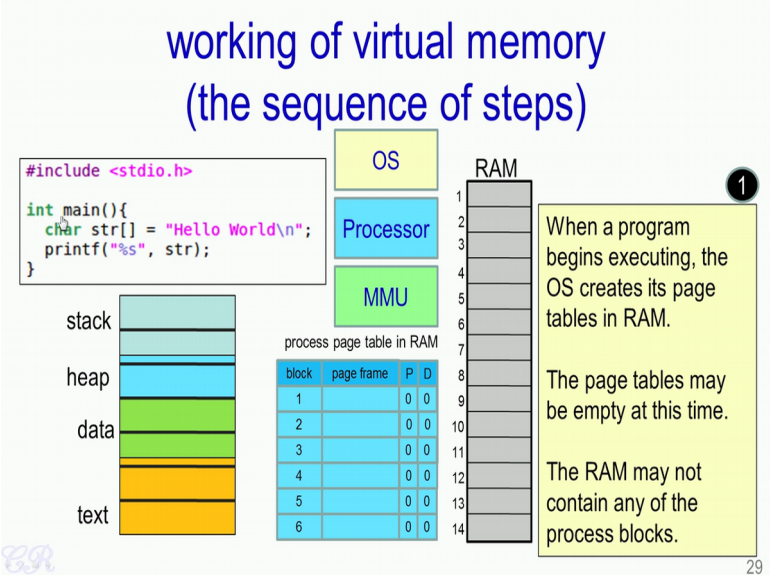
Now, corresponding to the virtual address what would happen is that the directory entry is used to index into something known as a Page directory. Now, this page directory is pointed to by a page directory pointer register present in the MMU. Now, the contents of this entry in the page directory would point to a particular page table. So, this (mentioned in above slide) page table is of 4KB. And, this is also of 4 kilobytes; that is, the directory is also of 4KB.

Next, the table entry is used to indexed into this (mentioned in above slide) particular 4KB table and from there, we obtain the first part of the physical address. And as usual, the second part of the physical address is taken from the offset. So, how does the scheme actually help us?

Now, what we see is that the number of page tables that can be present, that is the number of these (mentioned in above slide) is 2^{10} or 1024. And, how do we get 1024 is that we have a 10-bit directory entry over here and 2^{10} is 1024. Thus, the directory page table is of 4KB having 1024 entries, So 2^{10} entries. Since each entry points to a different table, thus we can have at most 2^{10} different page tables. Each entry in the directory would point to a different page table. Now, how is the scheme actually helping us?

So in total although we still have 4MB of page tables that is required, but the advantage we get is that they are not contiguous. So, we just need to have chunks of 4KB, which needs to be contiguous and this is easily obtained because each frame in the RAM is of 4KB. So, the 2 Level page translation would allow us to have non-contiguous page tables present.

(Refer Slide Time: 13:48)



Next, we will look at the sequence of steps that occur during the virtual memory mapping. So, let us start with the particular program (mentioned in above slide) and this is its virtual address space with different colors corresponding to each section of the program. Also, we have the various blocks and as we have mentioned, each of these blocks is of 4KB.

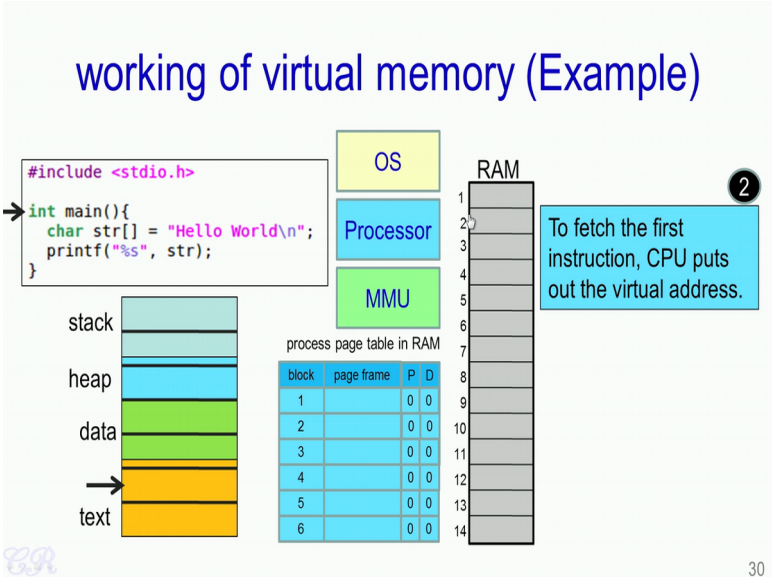
Then, what we have seen is that also there is a process page table present in the RAM, and there is the RAM itself which is again split into equal size blocks known as the Page frames. Now, there are actually three entities which are involved with the working of the virtual memory. That is the Operating System, which is the software component of this (mentioned in above slide) while the Processor and the MMU the memory management unit are the hardware components.

So let us say that we have started to execute this program (mentioned in above slide). And, this particular line i.e int main() in the program is the first instruction being executed. Now, let us see what is the sequence of steps that occur as the program executes. So, essentially when the user runs this particular program, it triggers the operating system to create this particular process page table (mentioned in above slide) in RAM. So, ideally this particular page table will have the present bits set to all 0 and the other parts of the page table may or may not be empty at this particular point in time.

Third, the RAM may or may not contain any of the process blocks corresponding to the newly created process.

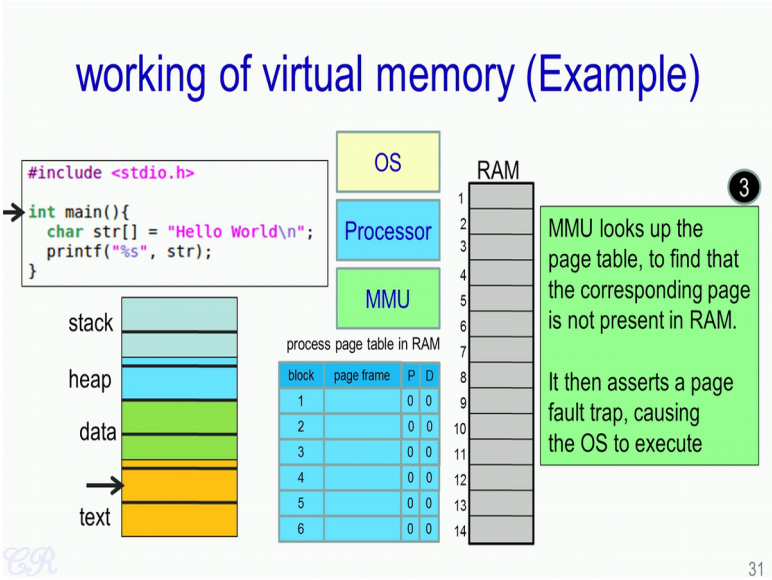
Next, what happens is that the operating system would transfer control into the main function of the program.

(Refer Slide Time: 15:50)



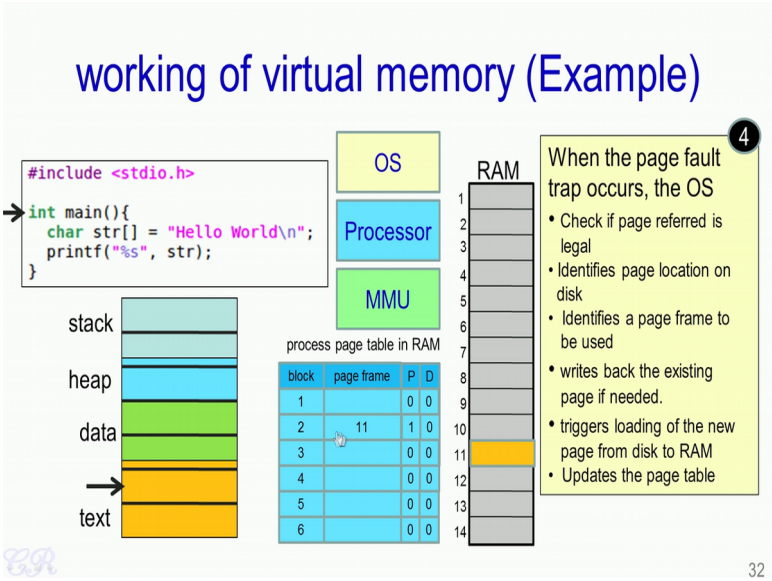
So this main function in the C code (mentioned in above slide) would probably point to a particular instruction in the virtual address space. So, this would result in the processor sending a particular virtual address corresponding to the main function. So, this virtual address would be sent on to the bus and this virtual address is then intercepted by the memory management unit.

(Refer Slide Time: 16:27)



And, the MMU would then look into the page table of the process and determine that, that corresponding page or that corresponding block is not present in the RAM. So, this is determined by the 0 bit present in this particular bit. So, that is determined by a 0 in the present bit (mentioned in above slide). So as a result of this, what the MMU is going to do is that it is going to cause a page fault to occur. So when a page fault occurs, it is going to trigger the operating system to execute.

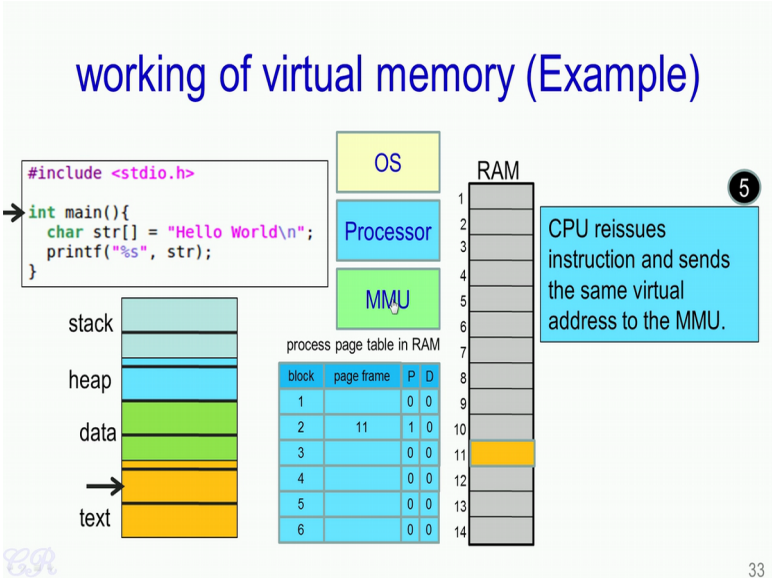
(Refer Slide Time: 17:12)



So then, the operating system is going to execute and determine the cause of the page fault. So, essentially there is a sequence of steps that the operating system does whenever the page fault occurs. So these are; that it will first check if the page referred is legal and only if it is legal it will continue. Then, it will identify the page location on the disk corresponding to this particular block (mentioned in above slide). Then, it will identify a page frame in the RAM that need to be used. Then, essentially if it requires that a swap out process is needed.

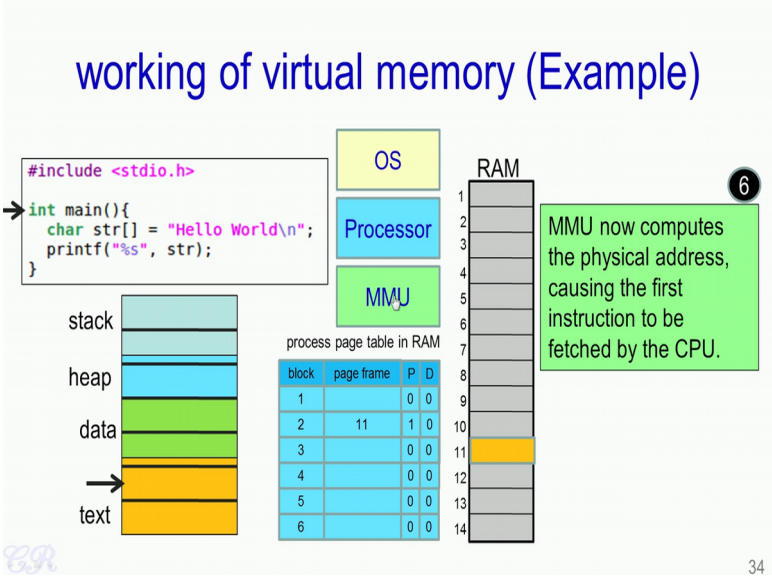
And if the dirty bit is set then the page which was previously present in the RAM would be returned back into the swap space. While this is followed by the block corresponding to this a virtual address being accessed loaded from the disk into the RAM. So, this essentially is done by what is known as a DMA transfer from the hard disk to the RAM. And, the operating system would just need to trigger this particular DMA transfer to be initiated and also, it is going to update the page table. So, essentially it is going to say that the first of all, the present bit is set to 1 and the page frame corresponding to block number 2 will be set to 11.

(Refer Slide Time: 19:02)



Then, after the page gets loaded into RAM, the transfer comes back to the CPU or the processor. And, the processor would reissue the instruction. So, this instruction would mean that there is a virtual address being sent on the processor bus and it would cause the MMU to convert that virtual address into the corresponding physical address.

(Refer Slide Time: 19:30)



So now what the MMU is going to see is that it is going to look into the process page table, is going to see that the present bit is set to 1. Therefore, it would mean that the block is loaded into a page frame in the RAM. And, it would then be able to convert the virtual address to the physical address and cause the instruction to be loaded from the RAM into the processor. And, in this way the processor executes an instruction.

So this is the sequence of steps that occur with the virtual addressing scheme. Essentially, it not only involves just the operating system, but there is a inter working between the operating system, the processor and the memory management unit in order to achieve virtual memory. In other words, in order to make the virtual memory begin to work.

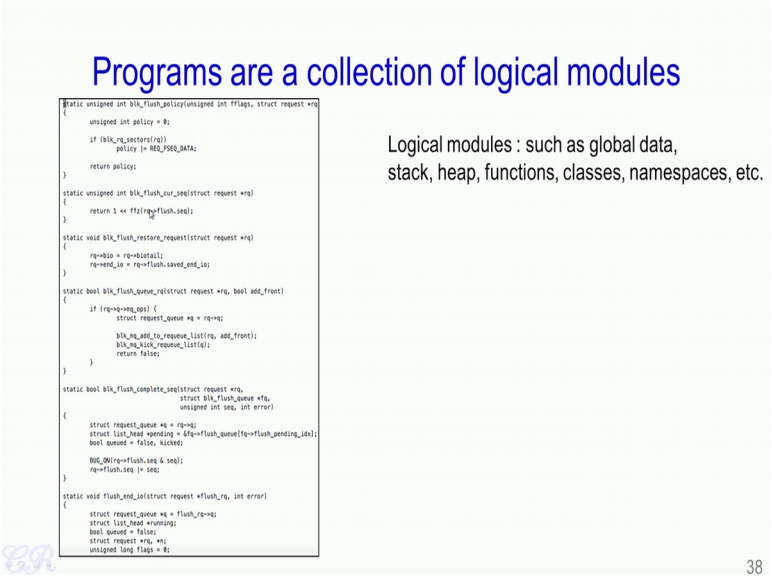
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 02
Lecture – 08
Segmentation

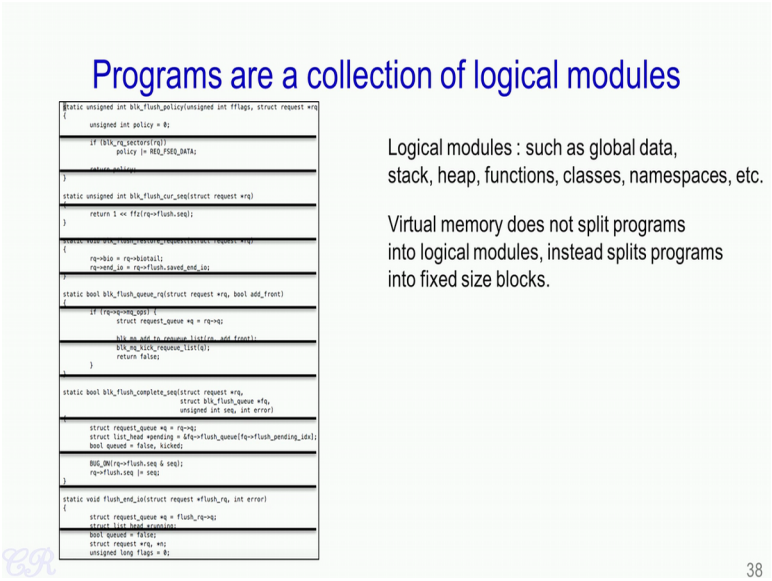
Hello. In the previous video we had looked about Memory Management, especially we had focused on a concept known as Virtual Memory. In this video we will look at another important concept for memory management which is known as Segmentation.

(Refer Slide Time: 00:33)



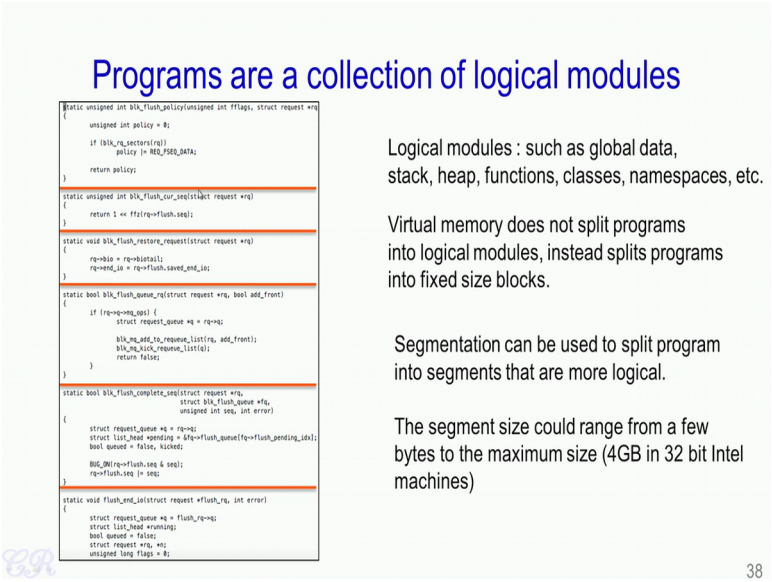
So, when we look at programs in general they can be split into Logical modules. So, logical modules for instance global data, stack, heap, functions, classes, namespaces, and so on.

(Refer Slide Time: 00:47)



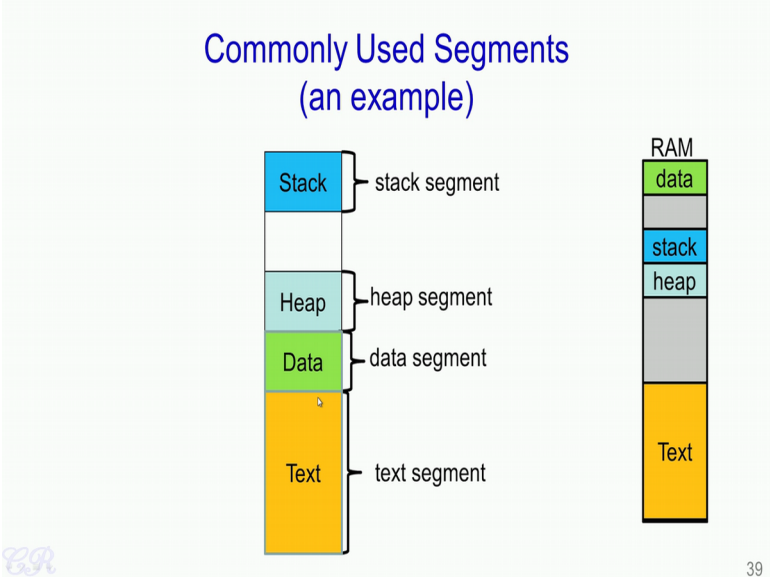
When we look at virtual memory on the other hand, virtual memory does not split programs into logical modules instead virtual memory actually splits programs into fixed sized blocks. So, while this would work in general it is not a very logical thing to do. For instance we may have a few instructions of a function in one logical block while the rest of the instructions of that function within a totally different logical block.

(Refer Slide Time: 01:17)



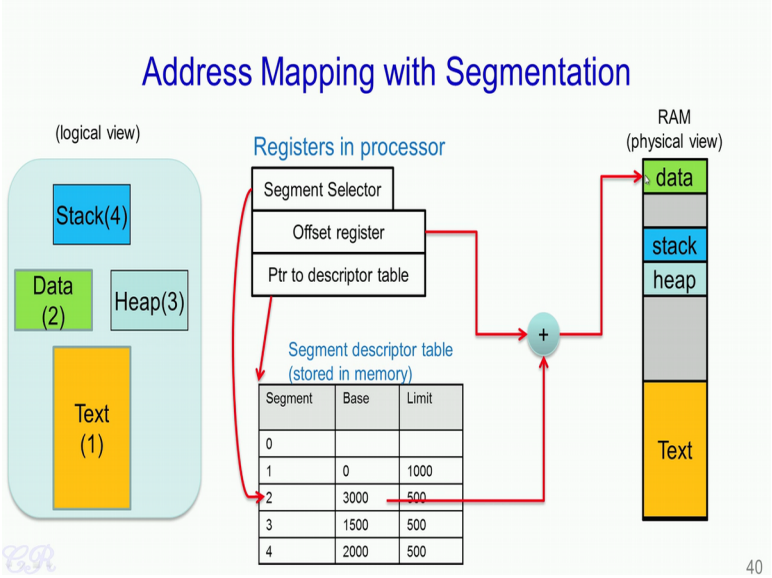
Segmentation on the other hand, achieves a more logical split of the program. So, we could define segments to vary in size from a few bytes to up to 4GB and we could define segments to be in a more logical order. For instance, we could have each function within our program to be in a different segment as shown in this particular slide (mentioned in above slide).

(Refer Slide Time: 01:39)



A very common usage of segmentation is to split the program into various segments such as the text segment, data segment, heap segment and the stack segment. So, this is known as the Logical view of the program.

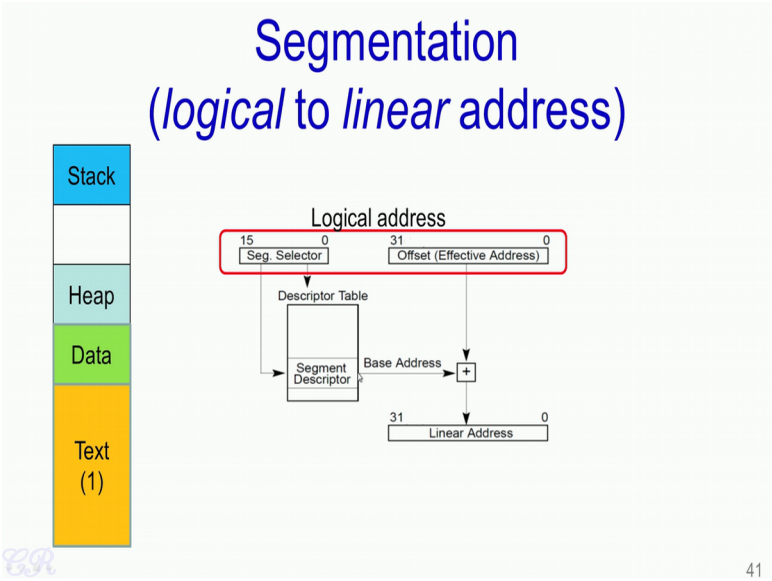
(Refer Slide Time: 01:55)



Now, let us look how the address mapping is done from the logical view to the physical view in segmentation. So, essentially we would have a segment descriptor table which is stored in memory. Each row in the segment descriptor table pertains to one particular segment. For instance, the Data segment 2 (mentioned in above slide) is at an offset 2in the Segment descriptor table, and the offset would specify the Base address in RAM and the Limit of the segment.

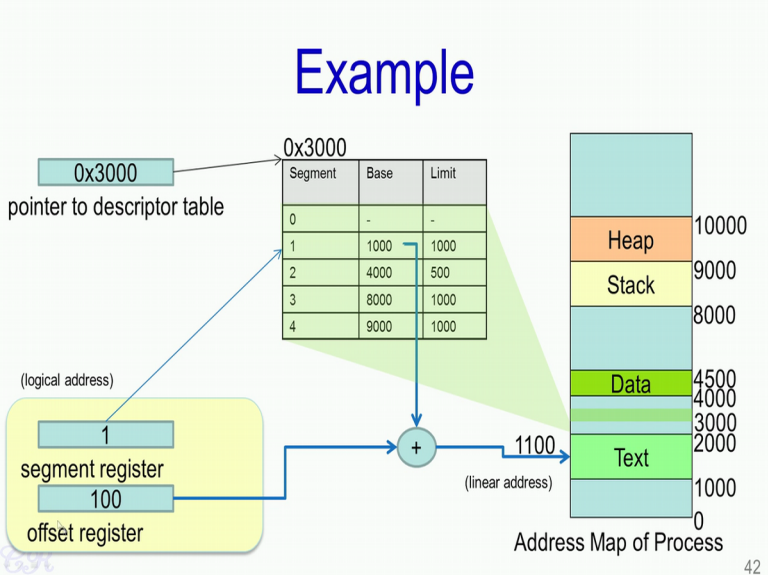
Now in order to make the mapping, the processor would have at least 3 registers that is the Segment selector, Offset register and the ptr to the descriptor table. So, as the name suggests the pointer to the descriptor table is a pointer to the memory location which holds the descriptor table. The segment selector on the other hand, is an offset into the descriptor table. The memory management unit in the processor would look up this particular offset (mentioned in above slide) and pick the base value 3000. So, this base value is then added with the contents of the offset register to get what is known as the Effective Address. So, this Effective Address will correspond to some address in the RAM.

(Refer Slide Time: 03:16)



Let us look at another view of this mapping scheme. The logical address comprises of two parts, it comprises of a segment selector as well as an offset address also known as the effective address. So, the segment selector in an Intel 32 bit processor is of 16 bits, while the effective address or the offset register is of 32 bits. So, what would happen is the contents of the segment selector is used as an offset into the descriptor table. The memory management unit would then pick up the base address from this particular offset (mentioned in above slide) and add the contents to the effective address to what is known as the Linear address.

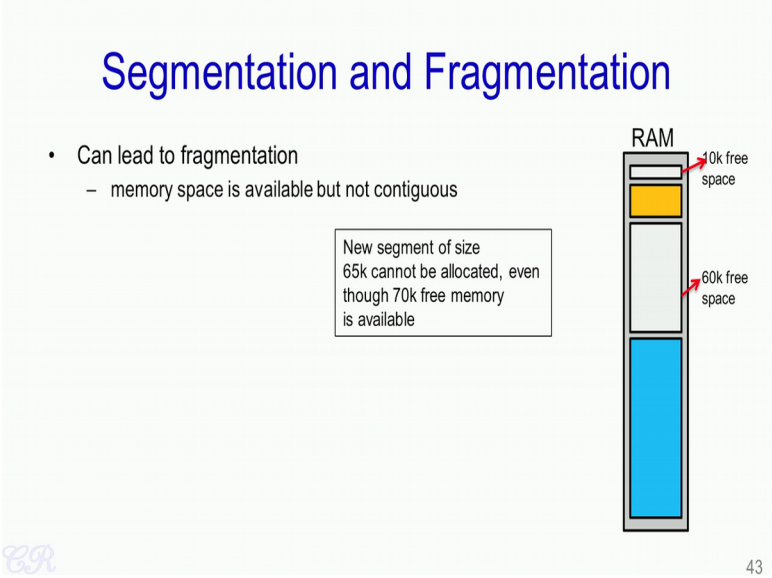
(Refer Slide Time: 03:58)



Let us look at the mapping done in segmentation with an example. Let us say the register containing the pointer to the descriptor table has a value of 3000. So, this means that at an address 3000 in RAM (mentioned in above slide) there is this descriptor table which contains the mapping for the various segments. Now let us say that the segment register has a value 1, so this means we are trying to use the segment at offset 1 in the descriptor table.

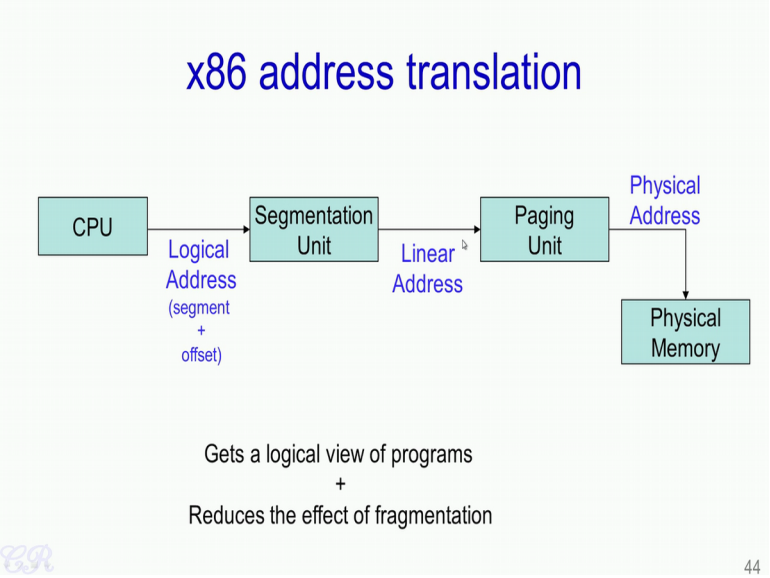
The memory management unit would then take the base address corresponding to this offset, which is 1000 in this case (mentioned in above slide) and use the offset register which has a value of 100 to get what is known as the linear address that is 1100. Now, the segment register along with the offset register form what is known as the Logical Address.

(Refer Slide Time: 04:51)



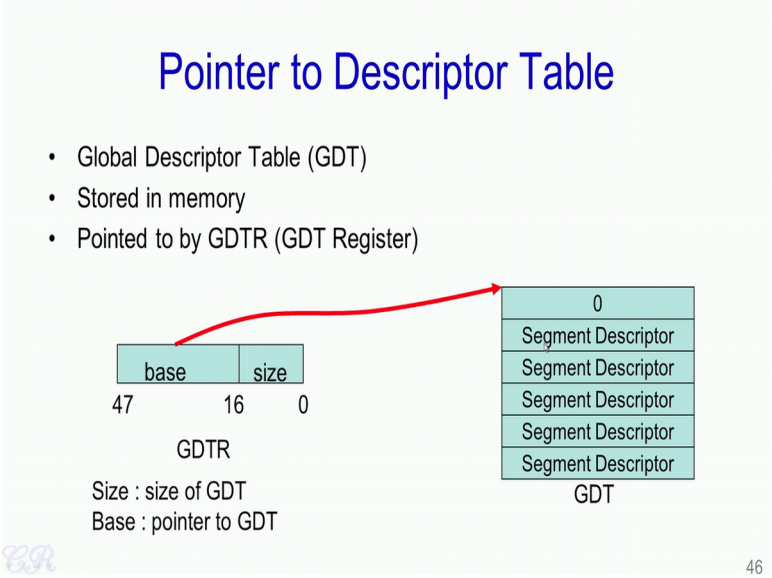
So, one of the biggest problems with segmentation is Fragmentation. Let us look at this particular example (mentioned in above slide); we have 70KB of space which is free in the RAM. However, the free memory is not in contiguous locations; we have 60 kilobytes of free space in one chunk, another 10 kilobytes of free space in another chunk. So, this cannot be used to allocate a new segment which is of 65 kilobytes. So, even though there is 70 kilobytes of free memory available, the memory is not in contiguous locations and therefore, cannot be used. Fragmentation is one of the biggest limitations of segmentation; however, fragmentation is much less an issue with virtual memory.

(Refer Slide Time: 05:41)



We will now look at how Intel x86 systems makes use of both segmentation as well as paging, so that advantages of both are obtained. So, in x86 system (mentioned in above slide), the CPU generates a logical address comprising of a segment plus an offset. So this is sent to a segmentation unit which then generates a linear address. The paging unit is essentially the virtual memory management which would take the linear address and generate the physical address. So, let us see how the segmentation unit is designed in x86 systems.

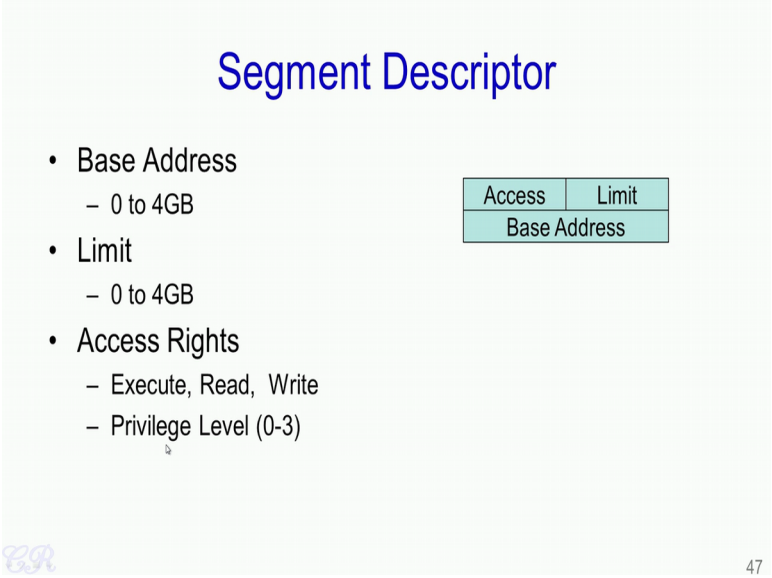
(Refer Slide Time: 06:16)



x86 systems have 2 types of descriptor tables. So, one is known as the Local Descriptor Table, while the other is known as the Global Descriptor Table which we will be presenting over here.

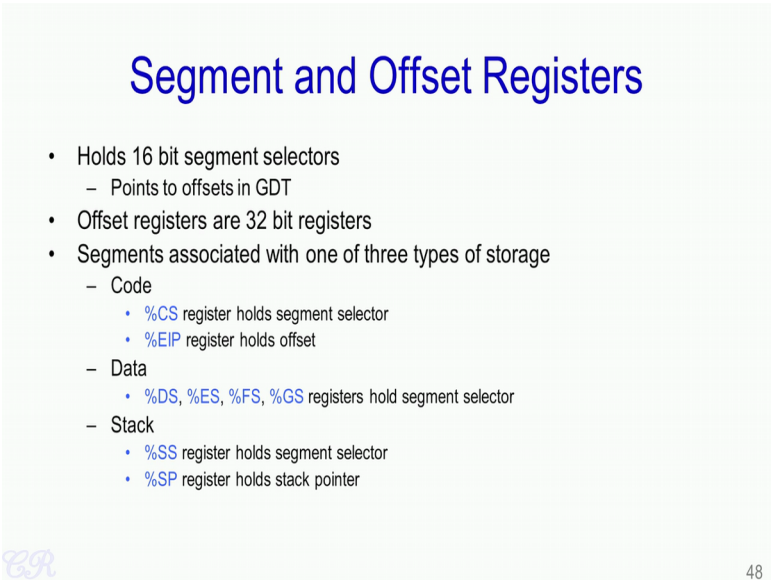
So the global descriptor table is stored in memory and has a format as shown over here (mentioned in above slide). So, essentially it has the first field which is 0, followed by Segment Descriptors. This particular global descriptor table is pointed to by a register known as the Global Descriptor Table Register or GDTR. The GDTR is a 48 bit register, having the following format. The least significant 16 bits contains the size of the GDT, while the upper bits contains the base address that is the pointer to the GDT; that is this pointer (mentioned in above slide). So, let us look at what the content of the segment descriptor is?

(Refer Slide Time: 07:06)



So, the segment descriptor contains 3 parts it has a Base Address, it has a Limit and it has an Access Rights (mentioned in above slide). The Base Address and the Limit can take values from 0 to 4GB, while the access rights are bits which specify various access policies such as Execute, Read, Write or the Privilege Level, for that particular segment.

(Refer Slide Time: 07:30)

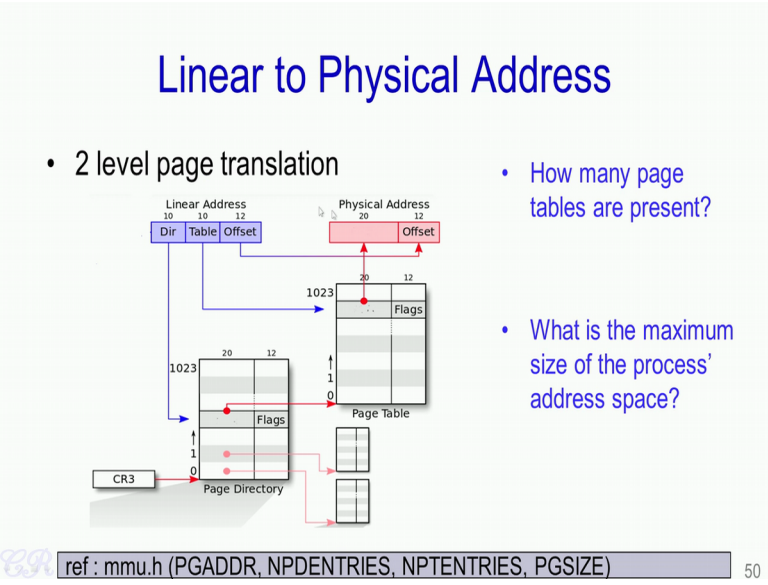


Next, let us look at the segment and offset registers in Intel 32 bit machines. The segment selector registers are 16 bit segment selectors which points to offsets in the GDT. The offset registers are 32 bit registers. So, quite often the segment selectors, a couple bit corresponding offset registers. For instance, in order to access the code segment we use the CS register which is the segment selector for the code segment, and the corresponding EIP register which is the offset register known as the Instruction Pointer.

In order to access the Data segment we have several segment registers such as the DS, ES, FS and GS. In order to access the Stack segment we have the SS register which holds the segment selector, and the SP register which holds the stack pointer. All these segment selector registers and offset registers along with the GDTR and the GDT table present in memory are used to convert the logical address to a corresponding linear address.

Next we will look at the paging unit which essentially manages the virtual memory mapping in the x86 system. So, the paging unit takes a linear address and converts that to an equivalent physical address which is then used to address the physical memory or the RAM.

(Refer Slide Time: 08:58)

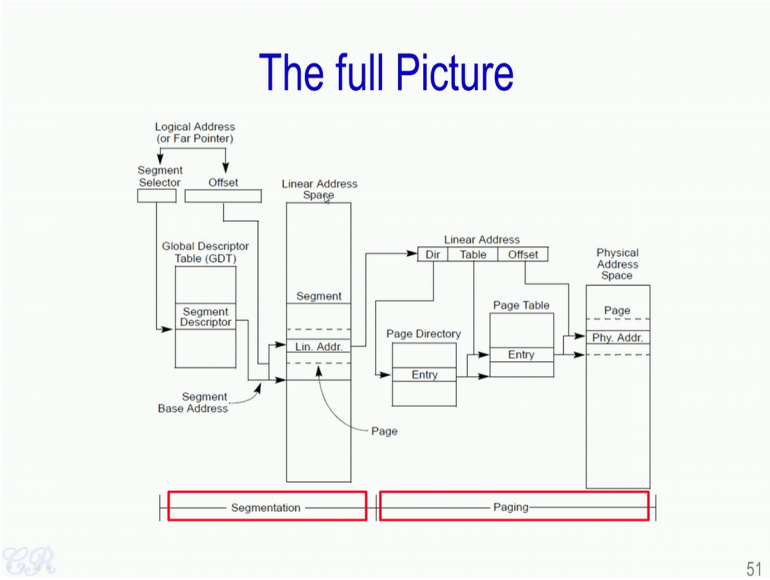


The paging unit in x86 system comprises of 2 level page translation. It takes a 32 bit

linear address which is split into 3 parts, the most significant 10 bits is known as the directory entry, while followed by which there is 10 bits for the table index and finally, the least significant 12 bits are the offset (mentioned in above slide). So the directory entry points to a particular offset in the page directory. The page directory is a special table which is present in the RAM and it is pointed to by the CR3 register. So the contents of the page directory point to a particular page table and an offset within that page table is taken from the table index. So the contents of this particular page table along with the offset are then used to form the physical address.

So I have two questions for you. So one is how many page tables are present? So how many of such page tables are present in a 32 bit Intel system? While the second question is what is the maximum size of the process's address space? So, given that each process has a, such a linear address to physical address mapping. So, I want you to actually find out what would be the maximum size of a process's address space.

(Refer Slide Time: 10:18)



This particular slide shows the full address translation in an x86 system. The CPU puts out a logical address comprising of a segment selector and an offset. The segment selector is an index into the global descriptor table into something known as the segment descriptor. The segment descriptor along with the offset then creates what is known as

the linear address, and this entire space is known as the Linear Address Map for the process. The linear address comprises of 3 components that is the Directory entry, the Table index and the Offset. So, the directory entry indexes into the page directory and this content is then you select a page table. The table index is used to get an offset within that page table and the contents of this, along with or the final 12 bits in the linear address is then used to obtain the final physical address which is used to read or write data to the RAM.

With these set of videos we had looked at memory management schemes such as virtual memory and segmentation and we have seen how Intel manages address translation in 32 bit systems.

Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

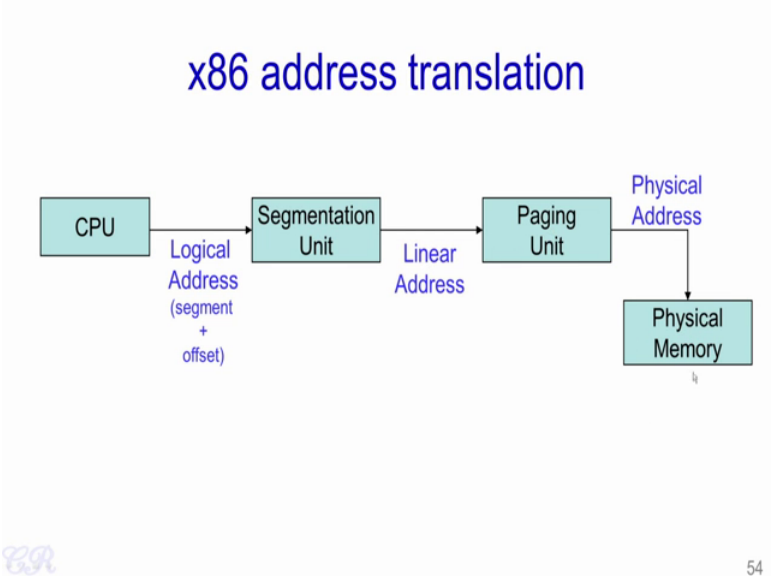
Week - 02
Lecture - 09
Memory Management in xv6

Hello. We had seen in previous videos about memory management schemes in Processors and about Virtual Memory and Segmentation, and we also had seen about how memory is managed in x86 systems.

(Refer Slide Time: 00:30)

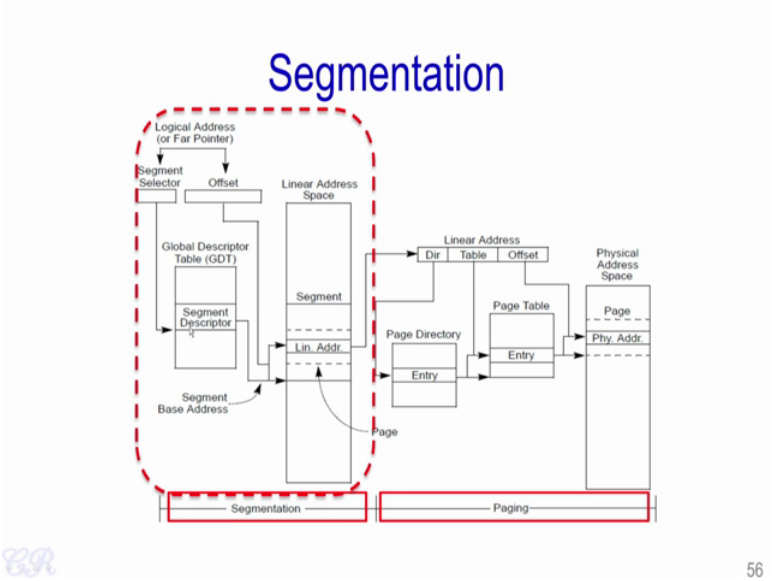
In this video we will look at Memory Management in xv6. So, xv6 is an operating system which is targeted for x86 platforms, therefore the video corresponding to memory management in x86 processors would be important. So, we would be referring a lot of xv6 source code in this particular video, therefore for reference you could actually look at the xv6 source code booklet (revision 8) which it can be downloaded from this particular (mentioned in above slide) website.

(Refer Slide Time: 01:05)



Now, just to recall in an x86 system, there is two levels of memory translation. So the CPU puts out a Logical address which comprises of a Segmentation + an offset. Then, there is a Segmentation unit which converts the Logical address into a Linear address and a Paging unit which converts a Linear address to the Physical address and only then the Physical memory or the RAM is actually accessed.

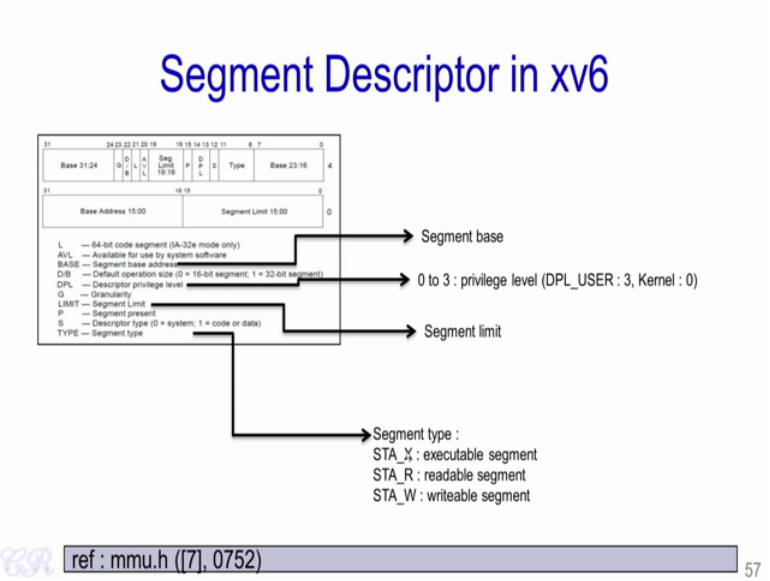
(Refer Slide Time: 01:36)



To get a full view of this memory management in xv6, we had seen this particular diagram (mentioned above). And in particular, we had seen about the segmentation unit followed by the paging unit. We had seen especially in the segmentation unit that there is a segment selector, which is a register such as the stack segment, code segment, the DS, ES, FS segment and this particular segment would index into a table known as the GDT that is the Global Descriptor Table.

And in this particular GDT is what is present the Segment Descriptor. The segment descriptor when combined with the offset will give you what is known as the Linear Address. Let us look more about how segmentation is handled in the xv6 operating system. So, first of all we will look more into the segment descriptor.

(Refer Slide Time: 02:38)

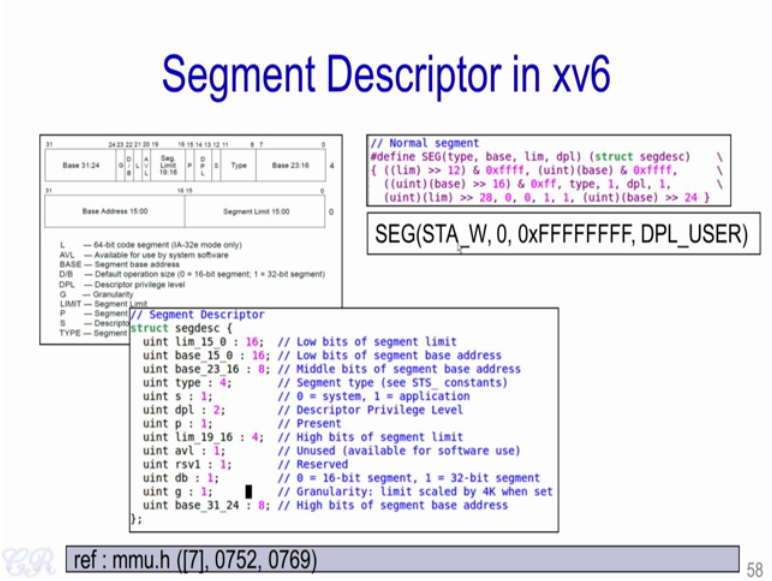


Each segment descriptor in an x86 system has 64 bits. So, these 64 bits can be viewed as 2 words of 32 bits each. The segment descriptor contains several attributes of the segment. Some of the important attributes are shown over here (mentioned in above slide). So, one important attribute is the segment base or the base address of the segment. So, this base address comprises of 3 parts; it is of 32 bits and comprises of 3 parts, 16 bits over here, bits 16 to 23 over here, and bits 24 to 31 present over here.

The segment limit contains the limits of the segment. So, this is of 20 bits and is present in 2 parts. You have 16 lowest significant bits of the present over here while the 4 most significant bits present here (mentioned in above image).

Another important attribute in the segment descriptor is the privilege level. The privilege level is of 2 bits and can have values from 0 to 3. So, user process's which have the least privilege level are given a value of 3, while operating system code which has the highest privilege level has a value of 0. Another attribute in the segment descriptor is the Segment type. So, these could have 3 values STA_X that is executable, readable segment as well as writeable segment. In addition to this, you could actually combine these attributes for a particular segment. For instance, you could have a segment, which is executable as well as readable. So, you could specify that the segment has a type STA_X as well as STA_R (mentioned in above slide).

(Refer Slide Time: 04:38)

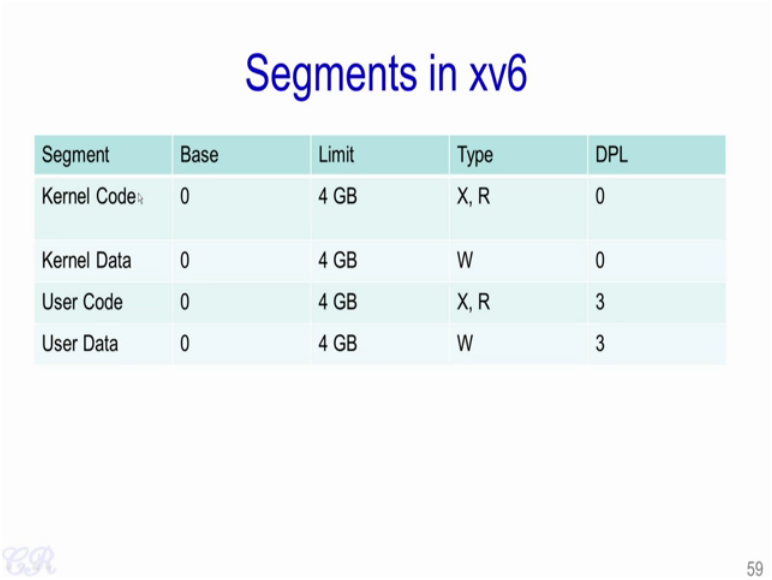


So In xv6, this segment descriptor is represented by a structure in the operating system code. So, if you look at mmu.h, you will actually see this particular structure (mentioned in above slide). So, as you can see all the attributes in the segment descriptor are represented in this structure. For instance, the base address which is of 32 bits and split into 3 parts is also represented by 3 parts over here. You have 16 bits over here followed

by 8 bits and the most significant 8 bits present over here. In addition to this, there is a macro in xv6 known as SEG which takes 4 parameters: type, base that is a base address, segment limit and the dpl. So, this macro is used to create a segment descriptor, it is a helper macro which is used to create a segment descriptor.

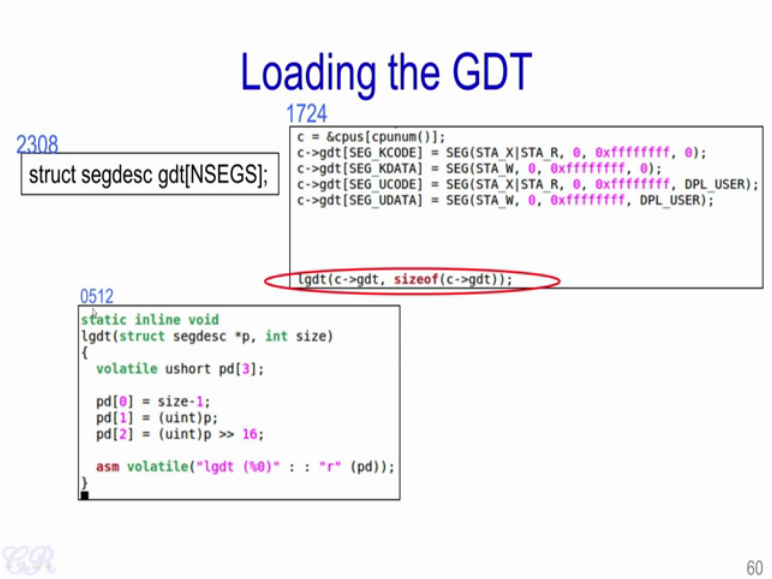
A typical usage of the SEG macro is as follows (mentioned in above slide). So, this particular usage of SEG creates a segment which has a base address of 0, it has a limit of 2^{32}-1 that is 0xFFFFFFFF. It is having a privilege level of user, specified by DPL_USER and it is of type W - that is the segment is writeable.

(Refer Slide Time: 06:00)



Xv6 does not make use of segmentation much, it only creates 4 segments. These are the Kernel code, Kernel data, User code and User data. All these segments have a base address of 0 and have a limit of 4 GB. All the code segments that is the Kernel code and the User code are of type executable and readable that is X, R (mentioned in above slide). The data that is the Kernel data and the User data are of type writeable that is W. Now the Kernel code and Data have a DPL value of 0, which indicates the highest privilege level while the User code and Data have a DPL value of 3 indicating the lowest privilege level. Next we will see how these 4 segments are created in xv6.

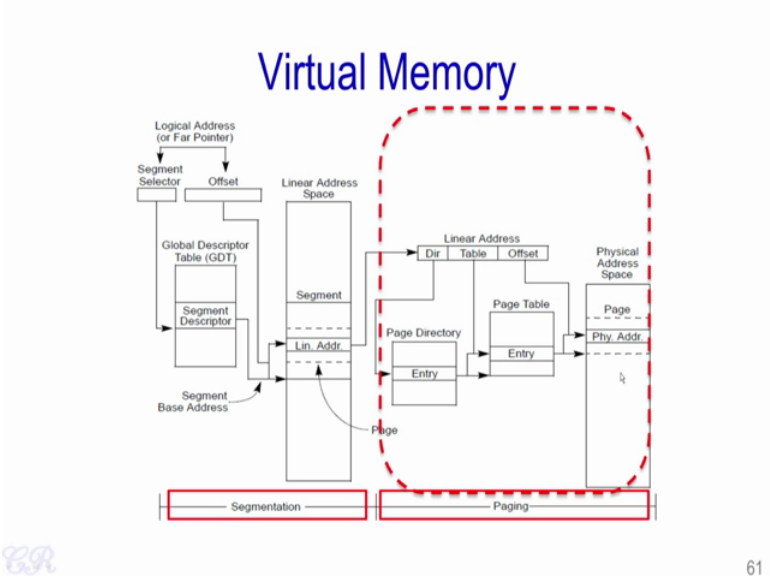
(Refer Slide Time: 06:54)



So, when it comes to segmentation it is required to have a GDT, which is present in memory. So this particular declaration which is struct segdesc gdt[NSEGS] which takes the number of segments is an array comprising the GDT. So If you look up line number 2308 of the kernel code, you will see this declaration. So, this particular memory array or this particular array is used to store the GDT table. So this particular array is filled by these codes (mentioned in above slide). So, we create the 4 segments as follows. So, we create the 4 segments has the Kernel code segment, the Kernel data segment, the User code segment and the User data segment. So, these segments are created according to the rules we have seen (in slide time 6:00).

Finally, we need to have the GDT register pointing to this particular GDT table, so that is done by this invocation of this particular function known as lgdt or load gdt. So, load gdt is a function present in line number 512; and essentially it takes two parameters, it takes a pointer to the segment descriptor which essentially is the pointer to this GDT table (C -> gdt) and the size of the GDT table which we have passed over here as size of c -> gdt that is the size of this GDT table. And, essentially it just going to invoke this particular instruction call lgdt (mention in above slide), which fills the GDT pointer in the mmu with the address of the gdt.

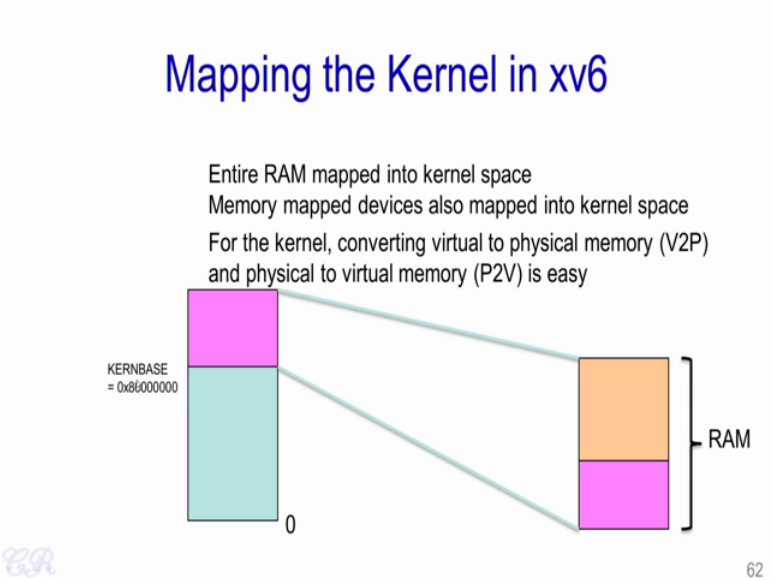
(Refer Slide Time: 08:38)



Next we will move on from segmentation to the paging unit that is how the virtual memory is managed in the xv6 operating systems. As we seen before in x86 systems, virtual address sync is actually managed by 2 levels of page directories. As a result, the linear address or the virtual address is split into 3 parts; you have a directory entry which comprises of 10 bits, a table entry comprising of 10 bits, and an offset comprising of 12 bits. The directory entry is used, to index into a page directory table which is present in memory.

In x86 systems, this page directory is pointed to by a register known as the CR 3 register. Now, the contents of the page directory entry is then used to point to a page table. So that the second part of the linear address that is a table index is then used to offset into the page table to get the page corresponding page table entry then the contents of the page table is then added to the offset to get the corresponding physical address.

(Refer Slide Time: 09:58)



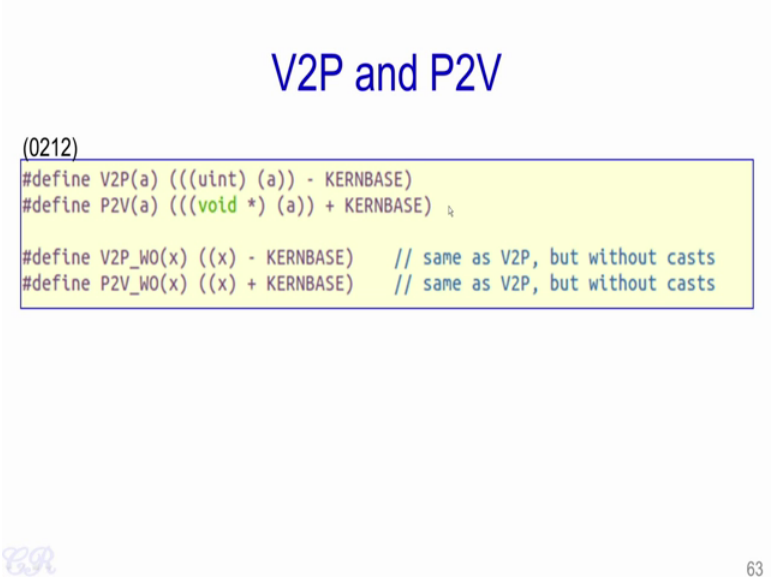
Now, we will see how this virtual addressing scheme is managed in xv6? As the xv6 begins to boot, the operating system Kernel code and Data gets copied into the lower regions of RAM as shown in this pink shaded region (mentioned in above slide). As the OS continues to boot page directories and page tables are created such that the entire RAM gets mapped into the higher region of the logical address space that is the 0th location of RAM gets mapped into what is defined as the KERNBASE that is 0x80000000. Similarly, every address in this RAM would have an identically mapped address in the logical space.

Why is such one to one mapping created between the logical address space for the Kernel and the physical RAM? The reason is that, which such a map the kernel could easily convert from a virtual address to the corresponding physical address that is given any virtual address in the kernel space that is some over here (mentioned in above slide). The operating system or the kernel could easily obtain the corresponding physical address using a macro known as the V2P or the virtual to physical memory conversion macro.

Similarly, in order to convert from a physical address to the corresponding virtual address would be very simple. So, any read any physical address in the RAM could be converted to the corresponding logical address or the corresponding virtual address by a

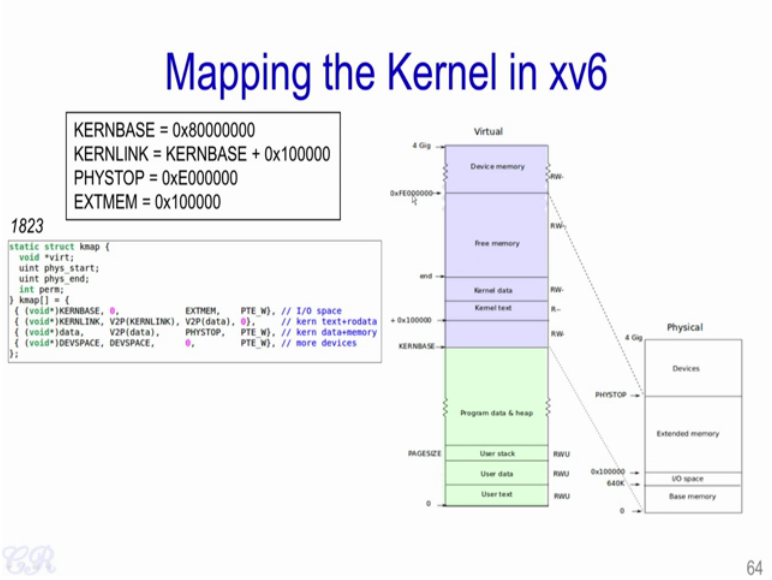
macro known as the physical to virtual memory macro is shortened as P2V. So the important thing during these in these 2 macros is this KERNBASE.

(Refer Slide Time: 12:01)



So let us see these two macros in more detail. So, if you look at these two macros, this is present in the line number 212 in the kernel source code that the macro P2V simply takes an address ‘a’ (mentioned in above slide)- which is a physical address and converts it to the virtual address by adding a KERNBASE. Similarly, the macro V2P takes a virtual address ‘a’, and converts it to a physical address by subtracting the KERNBASE. So, as you can see converting from virtual to physical and vice versa that is from physical to virtual becomes very simple from the operating system perspective.

(Refer Slide Time: 12:47)



Now let us see in more detail how the xv6 operating system creates the mapping between physical address and the logical address space. So, this particular figure here (mentioned in above slide) shows the physical RAM. So, as you can see the physical RAM is divided into several regions. The region from 0 to 640K is known as the base memory; the region from 640K to 1 MB that is 100000 is known as the I/O space. While the region from 1 MB to phystop - p h y s stop is the extended memory. Now this phystop is defined as a macro, which signifies the maximum amount of RAM present in the system. So, it could be something like 2 GB, 4 GB or 16 GB and could vary from system to system. So the region above this which extends up to 4 G is used by memory mapped devices.

Now, in order to create mappings for this particular physical RAM, the operating system would create page directories and page tables. So, in order to create this mapping, the operating system code defines several macros. So, some of the important ones are shown over here (mentioned in above slide). So, we have already seen about KERNBASE which is defined to be equal to 0x80000000 and there are other macros such as the KERNLINK, PHYSTOP and EXTMEM.

Now the KERNBASE determines the virtual address at which the kernel's space starts. So, if you actually look at this logical address space, you will see that the KERNBASE is defined at this particular location (mentioned in above slide). The region of memory below this KERNBASE location is user space and this is the region where user processes execute; while the region above this KERNBASE that is from KERNBASE up to the maximum of 4 Gigs is where the operating system is present as well as where it manages memory and interacts with devices.

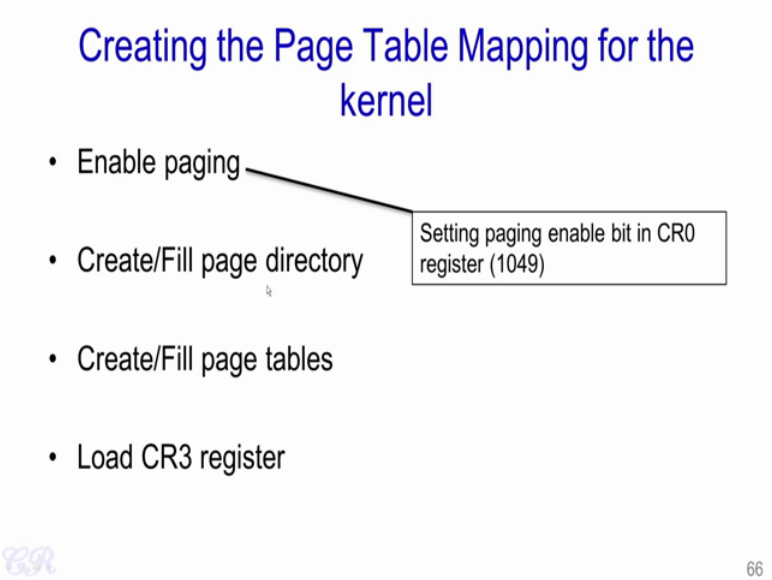
Now, another important macro is the KERNLINK, which is the KERNBASE + 1 MB. So, as you see over here (in above slide) this particular point, which is KERNBASE + 1 MB denotes the KERNLINK. So, this location is important because it is the location where the Kernel code and Data are present. So, starting at the KERNLINK, there is the Kernel text that is a Kernel code; and after the Kernel code is done, the locations comprise of the kernel data that is the global data which is present in the xv6 operating system. The end of the Kernel code and data is specified by this symbol 'end'. The region from end to this particular location 0xFE000000 is the free memory. This particular free memory is use for various things in the operating system such as for the OS heap as well as for allocating pages for user processes.

Now in order to create a map of this kernel space into the physical RAM a structure known as kmap is defined. So the kmap structure which is present in the line number 1823 contains four elements (mentioned in above slide). So, it has the *virt -, which is a pointer to the virtual address. Then it has the physical start address which indicates the physical address which gets mapped to the virtual address and the physical end that is the end of that region, and of course, there is the permissions. So, xv6 defines 4 such regions. So, we have the I/O space, the Kernel text and read only data, the Kernel data and memory regions and the device space. So, these four regions can be actually mapped into this particular kernel space.

For example, the I/O space which starts from the virtual address KERNBASE and gets mapped to the physical address 0 up to EXT EXTMEM. So, this is the 1 MB region starting from 0 to the end of the I/O space that is the 1 MB mag. So, this is known as the I/O space and is typically not used by the xv6 operating system.

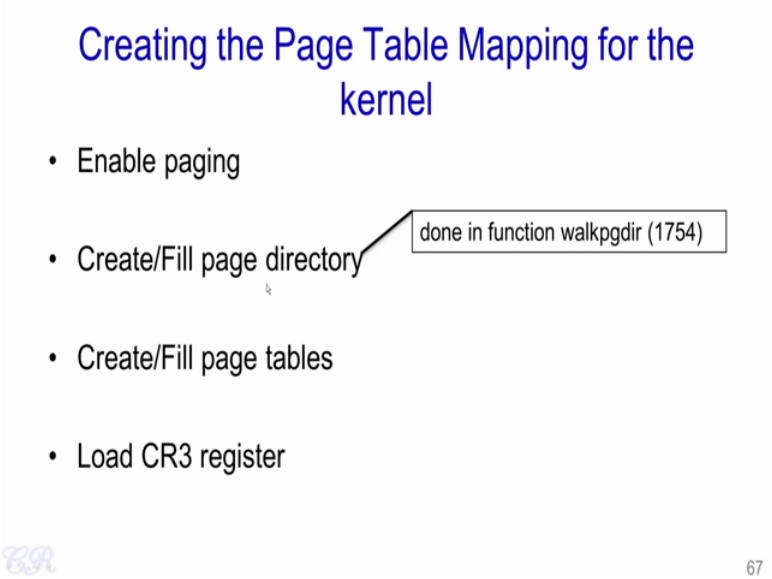
The region KERNLINK onwards is used to store the kern text + rodata that is Kernel code and read only data. So, KERNLINK region gets mapped from the start of the extended memory in the sense that KERNLINK gets mapped into the extended memory. So, it is at the start of the extended memory that contains the Kernel code and read only data in physical RAM. Then we have a region known as the data region which gets mapped into the free memory part, and finally, you have the device spaces which is above this particular location 0xFE000000.

(Refer Slide Time: 18:38)



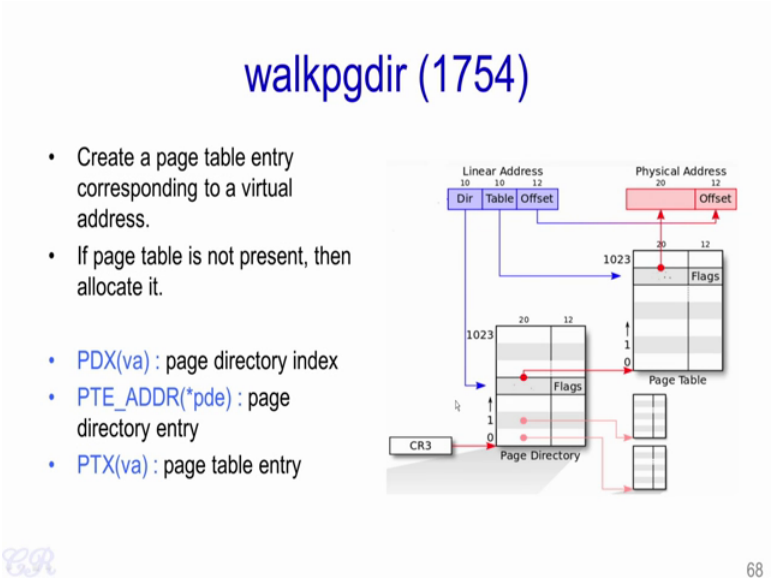
Let us see the sequence of steps that is involved in creating the mapping between the xv6 kernel space to the RAM. So, these are the four steps involved. First, there is a Enable paging. So, by default when the system is turned on paging is disabled. So, in order to turn on paging, a particular bit known as the paging enable bit which is present in the CR0 register has to be set to 1. So the CR0 register is a specific register in the x86 processor and in that register there is a bit called the paging enable bit which needs to be set to 1 to enable paging.

(Refer Slide Time: 19:25)



Another step is to create the page directories and the page tables. In order to create and fill the page directories, the function walk page directory or walkpgdir is invoked by the operating system. So, if you look up the source code, you will see this walkpgdir() function present in line number 1754.

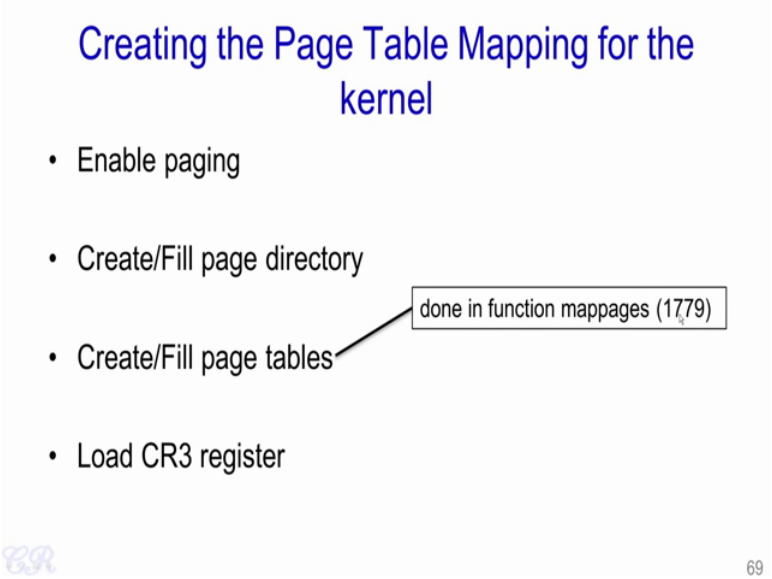
(Refer Slide Time: 19:45)



The walk page directory function creates the page table entry corresponding to a virtual address. So, essentially it is going to create an entry in this particular page directory. Secondly, if it finds that the corresponding page table entry in the page directory is not present then it creates in RAM a page table for it. So, this page table (mentioned in above slide) as we have seen will be of 1 page that is of 4KB. So, we have seen that there are 1024 entries and each entry is of 32 bits. So, in all this page table will be of 4 KB that is it will hold one page.

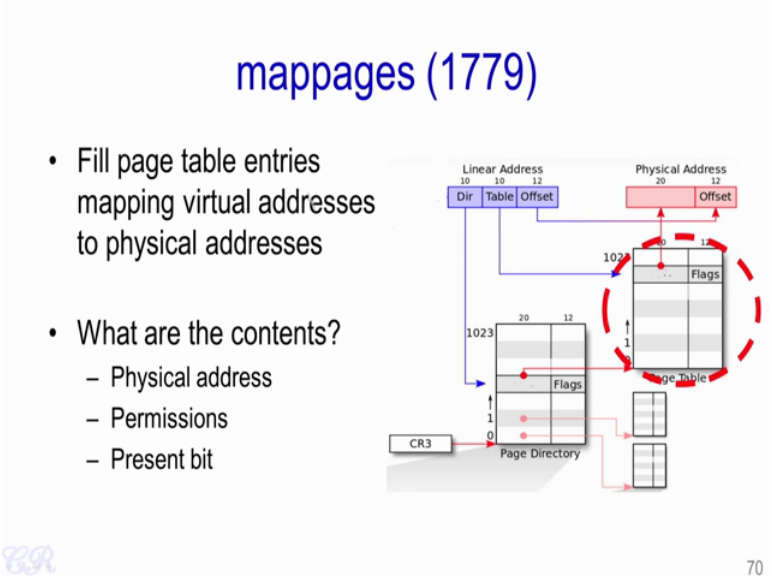
In a similar way, other page tables are created whenever required. So, walkpgdir function use a several macro such as the PDX macro which given the virtual address will extract the page directory index that is it is going to just give you this upper 10 bits of the linear address. Then we have the page table entry address which will give the page directory entry and the PTX macro which takes the virtual address and gives you the page table entry.

(Refer Slide Time: 21:06)



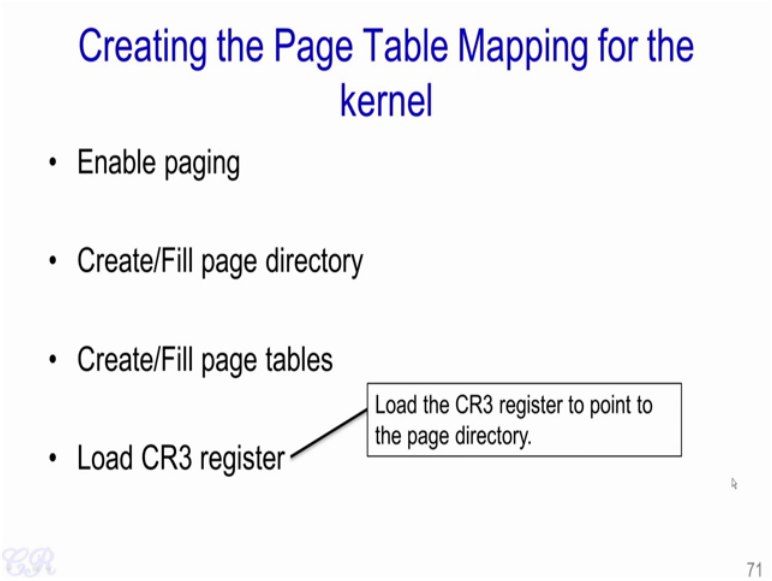
After creating the page directory, the next step is to fill in the page tables. So, this is done by the map page which is present in line number 1779 of the xv6 source code.

(Refer Slide Time: 21:17)



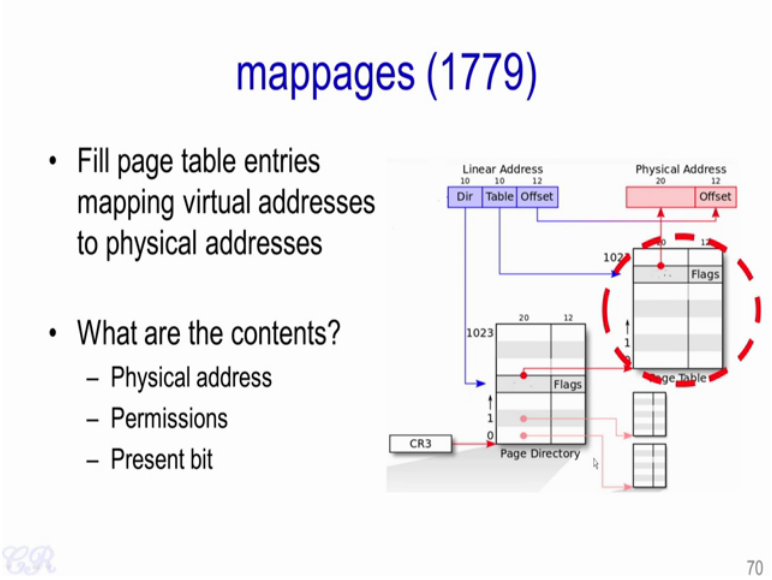
So, what the map pages does is that it is going to fill this particular page table with the mapping from the virtual address to physical address. So, this particular table entries (mentioned above in red circle) contain as we have seen the physical address mapping, permissions and also present bit. So, as we have seen that an entry in the page table is then used along with the offset to create the corresponding physical address.

(Refer Slide Time: 21:47)



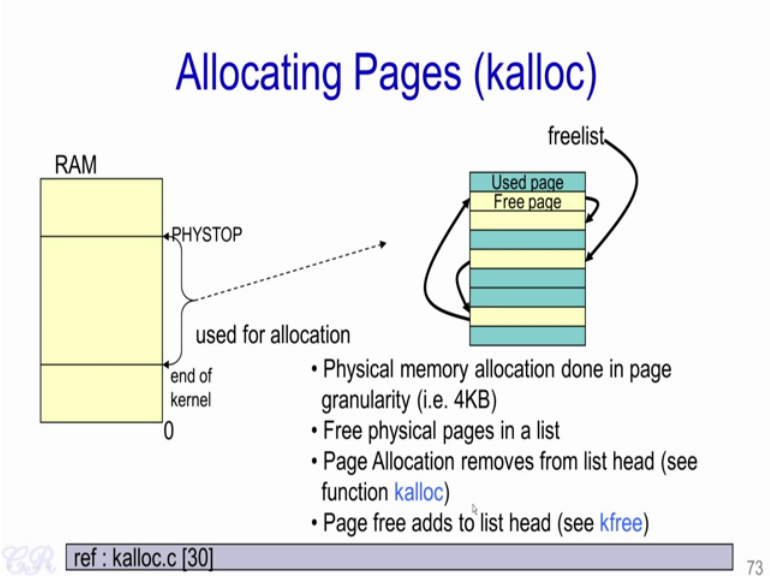
So, once we have created the page directories and the page tables, the final step is to load the CR3 register. The CR3 register is another register in the x86 processor which contains the pointer to the page directory.

(Refer Slide Time: 22:05)



So in other sense we have this CR3 register over here (mentioned in above slide), which is present in the processor of the MMU and it points to the memory location of the page directory. So now that we have seen how the xv6 code enables paging creates page directories as well as page tables. Let us see how the xv6 operating system allocates memory for various purposes. So, these purposes could be from allocating pages for user processes or for the operating system its use itself.

(Refer Slide Time: 22:45)



We had seen that the xv6 code and read only data gets loaded in the 0th location of RAM and extends upwards. At the end of the code and read only data up to PHYSTOP that is the physical end of the RAM is the free memory. This free memory is utilize by the OS for several purposes such as for allocating pages to user processes or for internal operating system book keeping requirements. So, what we will see now is how this free memory is managed by the operating system.

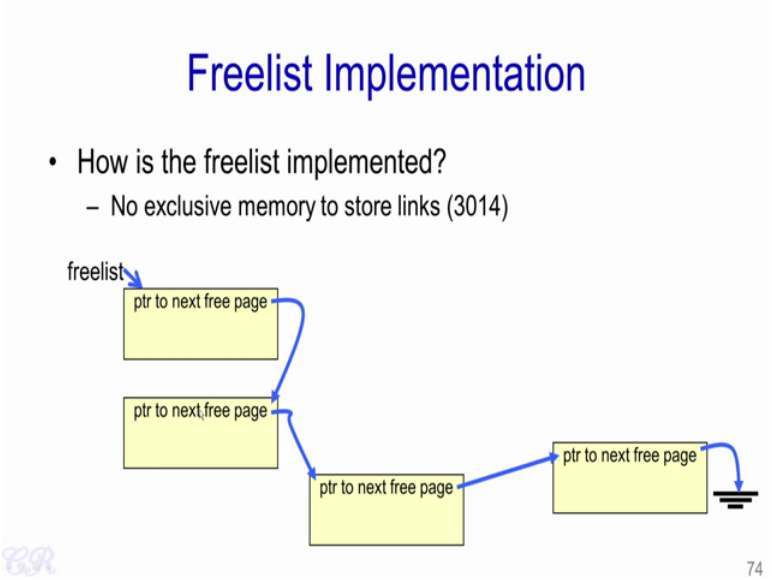
So essentially this free memory could be a large junk and it is split into pages of 4 kilo byte granularity. Thus we would have several pages present in this particular free memory region. Now, whenever required that is on demand a page would be allocated to a particular calling function and used for it is requirements. So, after the usage the particular page is freed.

Now, the next thing that we need to think about is how the operating system or that is how the xv6 OS determines which page to be allocated and how the pages should be freed? In order to do this the xv6 code maintains a link list of free pages. So, as you can see here (figure in above slide), this particular figure has pages which are either blue or yellow. So, this entire region corresponds to the free memory region in the RAM, the

blue region are the pages which is utilized by either a user process or by the operating system itself while the yellow pages are the once which are free.

Now, all the free pages are linked together using a link list which is pointed to by a pointer known as the freelist. So the freelist points to the head of the link list and all the free pages are linked together with this list. Now in order to allocate a page the kalloc function is utilized. The kalloc function would essentially remove a free page from the list and assign that page to be used. In order to free a page the kfree function is used; essentially the kfree function would add the free page back into the list.

(Refer Slide Time: 25:15)



So the link list is as shown over here (mentioned in above slide) where you have the freelist pointer which points to the head of the list and then there are pointers to the consecutive free pages. Now, one thing which is different from the standard link list is that there is no exclusive memory to store the pointer to the next page. Note that each of these pages is of 4KB and this 4KB free pages is not used for any other data thus the 0^{th} location in this page contains the pointer to the next page.

In a similar way, the 0^{th} location in this page would contain the pointer to the next page and so on. Thus we are creating this list. So the freelist points to the first or the head

node of the list and then the 0th location of that node points to the consecutive page and so on until we reach the end of the list. So, with that we come to the end of how xv6 manages memory. So, it is not the best management scheme that is possible, but it is a representative of what several operating systems actually do to manage memory.

Thank you.

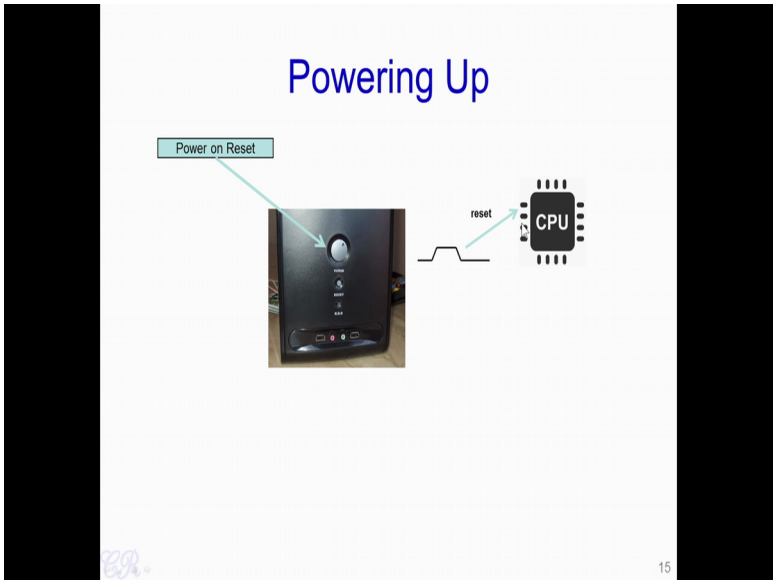
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 02
Lecture - 10
PC Booting

Hello. In this video, we will look at how a PC Boots, right up from you turn it on to the time the operating system executes. Now this particular video is especially applicable for Intel and AMD based platforms. So, one big thing which we should remember when we are talking about the Intel platforms that we typically use in desktops or laptops is the concept of backward compatibility.

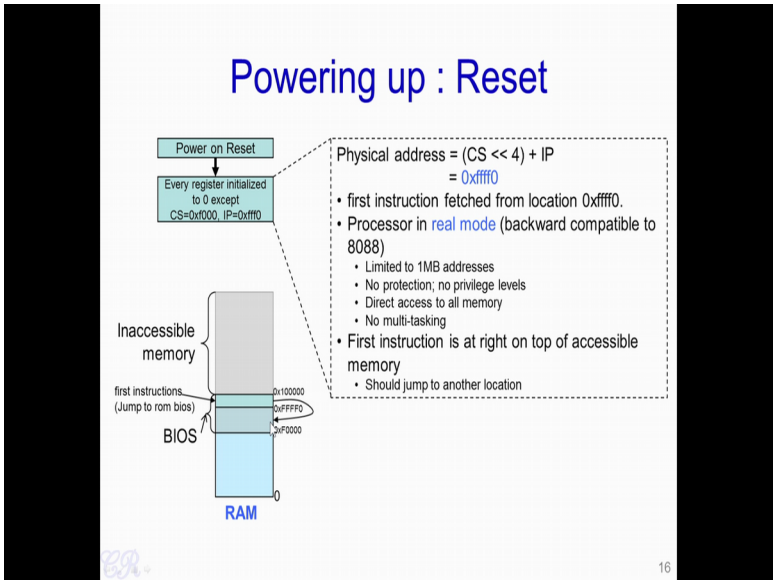
So as we have seen in a previous lecture. So Intel maintained backward compatibility. So, it ensures even today that any code which was developed on an Intel processor 20 or 30 years back which still execute on the an Intel processor are used today. So because of backward compatibility a lot of things that we actually do while loading an operating system would reflect on what was done 30 years back. Essentially, the things what happened in and a system in around 95 or 97. That is when the 386 based processors where used are still done today on the latest i7 processor from Intel. So, we will look at how PC Boots.

(Refer Slide Time: 01:50)



Now, we all know that in order to start a computer, we need to press the reset button or the start button present in the desktop. So, what actually happens internally is that when you press this button it is going to send a signal to the CPU. The signal for instance would be a pulse, these an electrical pulse which gets created when you press the start button or the reset button. And this pulse is sent to a specific pin on the CPU known as the reset pin and when the CPU obtains or gets this particular signal about the reset, it is going to start booting. So let us see what are the various steps involved when the CPU starts to boot?

(Refer Slide Time: 02:46)

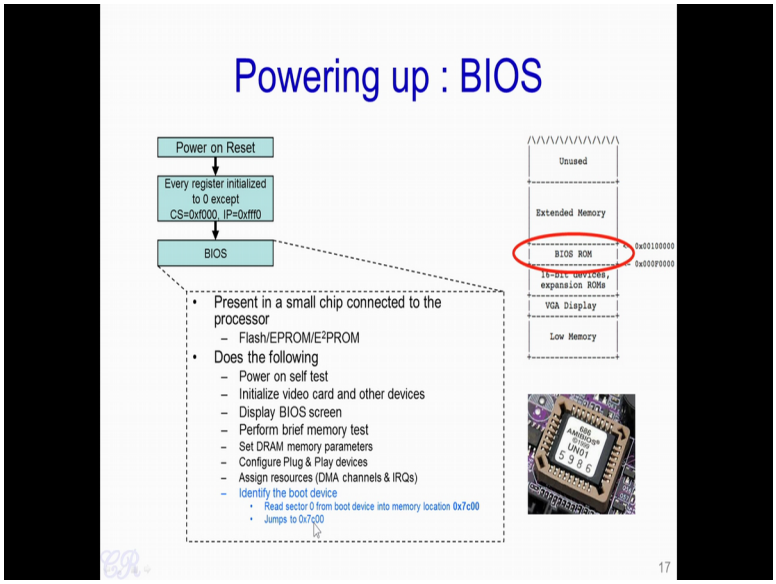


So, we had seen the power on reset. Now when the power on reset comes and the CPU detects it, what happens is that every CPU register which is present inside the CPU is initialize to 0 except for 2 registers and these registers are the code segment and the IP. Now when the reset occurs the code segment is set to the value of 0xf000 and the instruction pointer is set to 0xfff0. So, if you recollect how an 8088 or an 8086 processor computes its address, is going to take the code segment register in this case f000 shift it by 4 bits and add the instruction pointer.

As a result the physical address or the address for the first instruction to be executed will be present in 0xffff0. Now if you look up the RAM module, and which we had covered in the previous videos, what we would see is that the memory address corresponding to 0xffff0 would be pertaining to the BIOS area. In fact, this particular memory location 0xfff0 is just 16 bytes below the 1 MB mark. So, this particular thing 0x100000 that is 1 MB is this particular point or this particular address and the first physical address that is put on the address bus by the CPU is 0xffff0 which is 16 bytes below 1 MB mark. So, as a result if you want to boot your system, it should be ensure that at 0xfff0 memory location we have a valid instruction which is present. So, this point over here (mentioned in above slide) is the first instruction that is present.

Now, another thing what happens is that soon as the power is reset, the processor in is set to what is known as the real mode. So, in the real mode, the processor is in a backward compatibility mode with the 8088 or the 8086. So, recollect that the 8088 or 8086 processor could address at most 1 MB. So, it could address at most 0x100000 and this is shown as the green region in this RAM. There were other features of the real mode; they were no protection, no privilege levels, direct access to all memory and no multi tasking. So, the first thing that the instructions in this particular location 0xffff0 should do is, to jump to a other location. So, essentially what would happen is that it would jump to a location in the BIOS and this jump would trigger the BIOS to start to execute.

(Refer Slide Time: 06:23)



So next, we have that the BIOS ROM (mentioned in red circle) would begin to execute. So, as we know the BIOS is the basic input output system and it is a read only memory often these days it is in the form of a Flash or is E^2PROM and you would actually notice a particular chip (mentioned in above slide) which is present on your system. So some CPU's also display that particular BIOS name while booting up. For example; this particular chip is the AMIBIOS and it may get displayed by the system boots up.

So the BIOS present in this particular area of the RAM will begin to execute code in real mode. So, the BIOS does the following; essentially 1st does a power on self test where it

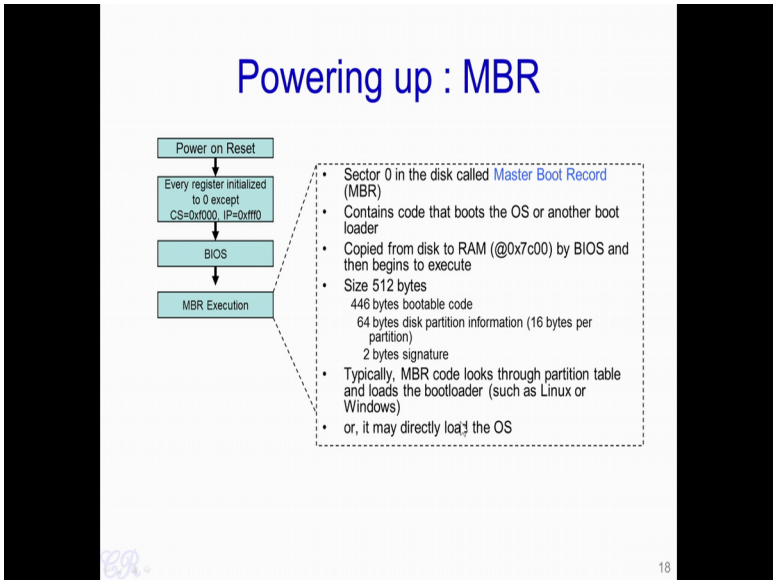
checks the system for correctness. It ensures that all parts of the system are working properly. Next it initializes the video card and all other devices which are connected to the system then, optionally it may display a BIOS screen on the monitor. Note that we have initialized the video card, and therefor the monitor is activated and it is capable of displaying things. So, the BIOS screen will now be able to display on the screen.

Then it performs what is known as a memory test and sometimes some of the BIOSs also determine what memory is used and also the amount of memory that is present in the system. After the memory test, some parameters are set. For example, these correspond to DRAM parameters and the BIOS will ensure that various requirements of the DRAM such as the frequency at which the DRAM capacitors are refreshed are set adequately.

Then plug and play devices are configured, in the sense that all devices which are plug and play are going to be queried and the BIOS will then determine how much memory is required for each of these devices, and these devices are then allocated memory in the system. After that the BIOS with assign resources to DMA channels and the various IRQ's that is the interrupt request. From our perspective what is important is the next step where the BIOS identifies the boot device, that is the device which most likely would hold the operating system. It would read the sector 0 from that boot device into the memory location 0x7c00.

So note that, 0x7c00 is a memory location in the low memory region of the RAM. So, what it would do is, it would copy the sector 0 which is typically of 512 bytes from the boot device which is typically the hard disk into the memory location 0x7c00. So, at the location 0x7c00 we would have 512 bytes of code which would help in booting the operating system. The BIOS then causes a jump to 0x7c00 what it means, is that the code present in location 0x7c00 which is present in the low memory of the RAM will began to execute.

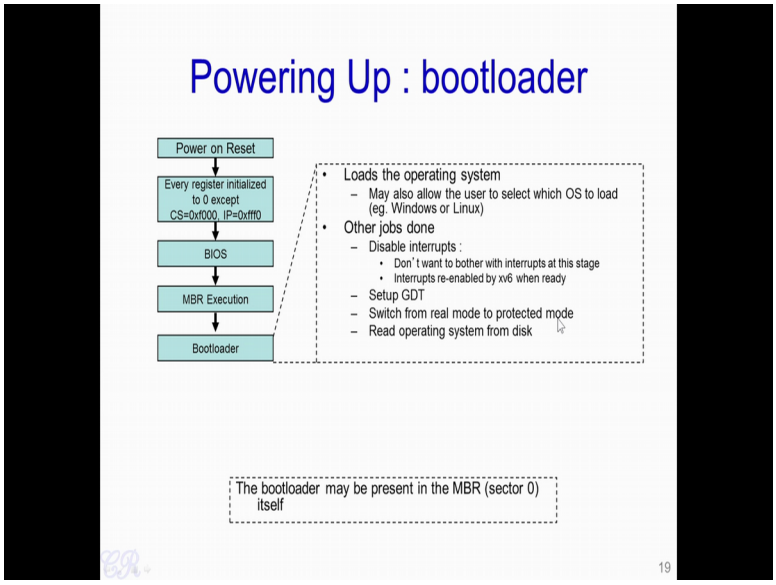
(Refer Slide Time: 10:21)



Now this memory, the memory present in 0x7c00 and a copied from sector 0 of the hard disk into the RAM, is known as the MBR or Master Boot Record. So, this particular code is of 512 bytes out of which 446 bytes are instructions and contain bootable code - about how to boot the system. There are 64 bytes which have information about the various partitions that are present on the disk. Essentially the 64 bytes are divided into 16 bytes per partition and then there are 2 bytes of a signature which is used to identify whether this is in fact an MBR code.

So, this code begins to execute from the location 0x7c00 present in the RAM and what it typically does is that it is going to look into the partition table which is present, and it is going to try to boot the operating system. So, essentially in order to do this, it first loads what is known as the boot loader of the operating system. So, each operating system would have its own boot loader. For instance LINUX would have its own boot loader or windows would have its own boot loader and so on. And, optionally it may directly load the operating system by itself. So, we will see what happens in the boot loader.

(Refer Slide Time: 12:04)

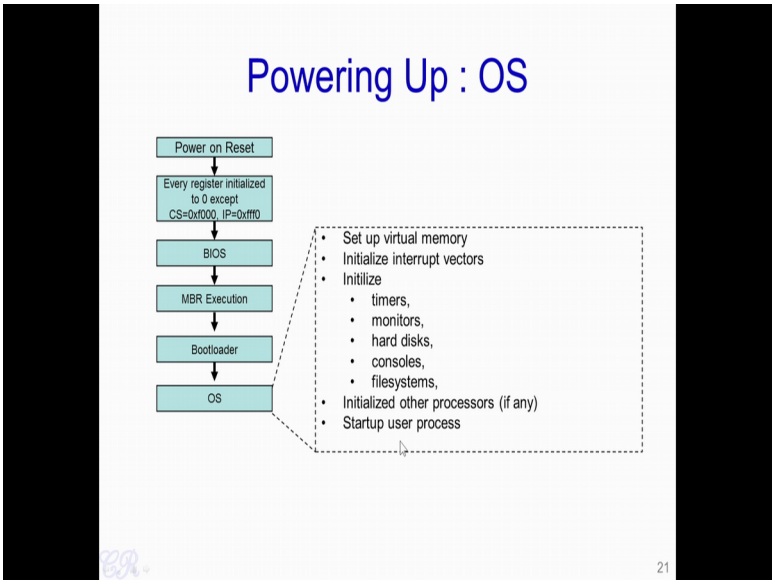


So, after the MBR executes, the boot loader would executes. So, the main job of the boot loader is that it loads the operating system. So, it optionally like some operating systems that we see today, it may give an option to the user to select what operating system to load. The other jobs that are done by the boot loader is to disable interrupts, set up the GDT switch from real mode to protected mode and read the operating system from the disk into the RAM. So, these are things which are done by the xv6 operating system. So, there may be slight variations when we go from one boot loader for one operating system to another operating system.

Sometimes what may happen is that we do not have this MBR code present at all. In such a case the BIOS or rather in such a case the boot loader itself is present in the sector 0 of the hard disk and the BIOS will load the boot loader into the location 0x7c00 and jump to the boot loader. Essentially what is happening is we are skipping this particular MBR execution. So, once the boot loader executes and sets up the processor and the GDT and switching from real mode to protected mode, it would load the operating system from the disk.

Now the protected mode is a 32 bit mode, essentially where we extent the memory region that can be accessed from 1 MB to the entire region of 4 Giga Bytes. So, we will not go into more details about how this particular protected mode is activated so on.

(Refer Slide Time: 14:13)

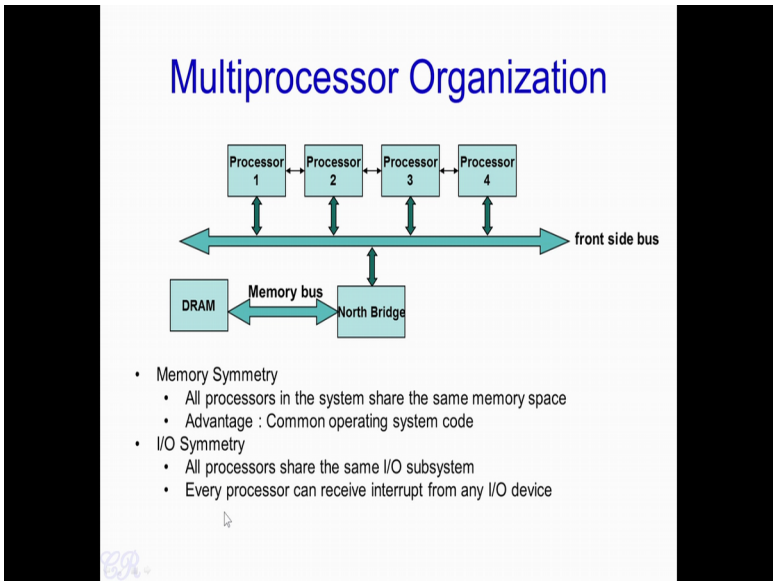


So, once the boot loader loads the operating system, it then transfers control into the operating system. And the operating system does several things, like it sets up virtual memory. So, this includes setting up page directories and page tables so on. It initializes interrupt vectors and the IDT interrupt descriptor tables and the other aspects pertaining to interrupts. Then it initializes various devices present in the system like timers, monitors, hard disks, consoles, file systems and so on.

Then it may also initialize other processes if they are present, and finally it would startup the first user process. So, in a future video, we will see what the first user process is? So this particular user process (last point in above slide) is the first process that executes in user's space. So, you may recollect that all of this (all points mentioned in square box) executes in the operating system and essentially this executes in the Kernel Space and it is only at this particular point during the boot up sequence, will the user process start to execute.

So, after this, what is expected is that this first user process will spon various jobs or user process jobs, various DRAMs and so on; and one of its jobs is to create a shell. So, this shell would be then used by the user, to run various programs and commands and so on.

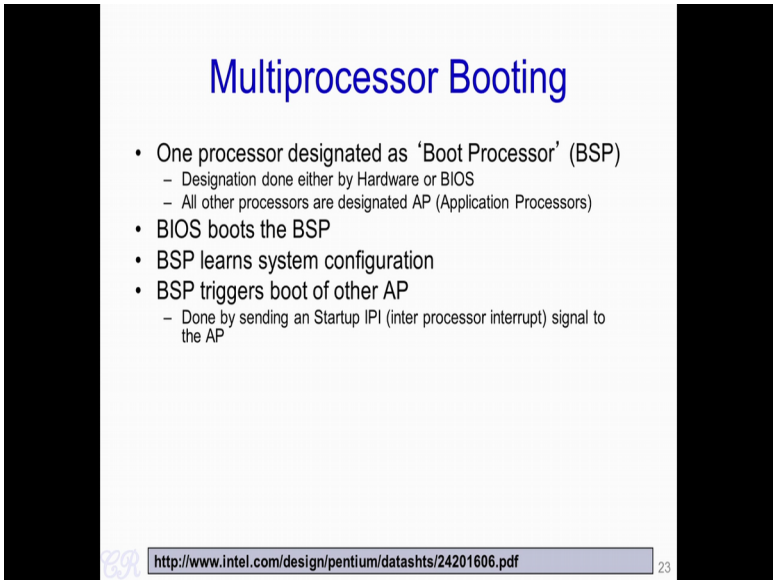
(Refer Slide Time: 16:05)



So, we will now look at systems which have a multiprocessor present in them. So, as we have seen in a previous video, so in the Intel type of architecture which have a multiprocessor present. So, the all processors share a front side bus and on the front side bus there is a chip set or the north bridge which interfaces with the memory bus. So, essentially in this Intel type of architecture we have memory symmetry. So, what this means is that, all processors in the system share the same memory space.

Essentially in order to access a particular DRAM location, all processors would need to send the same address to the DRAM, and the advantage of having such a symmetric view of the memory is that we can have a common operating system code which could execute in any of these processors. Similarly, there is what is known as the I/O symmetry. Essentially what this means is that all processors share the same I/O subsystem; essentially all processors can receive interrupts from any I/O device.

(Refer Slide Time: 17:30)



Now, in order to boot a multiprocessor system, what is generally done is that one processor is designated as the 'Boot Processor' or the BSP. So, this designation is done either by setting a particular signal in the hardware or by the BIOS itself, and all other processors are designated as 'Application Processors'. So, when the system is powered on, it is only the boot processor which begins to execute. So, the BIOS will execute in the boot processor and that is the BSP, and the BSP then learns about the system configuration.

It determines how many other APs are there, that is how many other application processors are present in the system and then it triggers the booting of the application processor. So, after it does all the required initialization it would trigger the boot of the application processors. So, this triggering of the boot is done by something known as the startup IPI or the startup inter processor interrupt.

This is a signal from the BSP that is the boot processor to the application processor. So, when the application processors see this signal they will begin to boot and of course, they identify that they are not the main BSP, but rather the application processor. So, they skip various aspects such as initializing the various devices present in the system and so on. In this video we had seen how the CPU boots right from the time the power on reset

is provided to the processor to the point when the operating system begins to execute and spons the first user process.

In the later part of this course we would see various aspects about how the operating system manages memory and manages different processes which are running in the system.

Thank you.