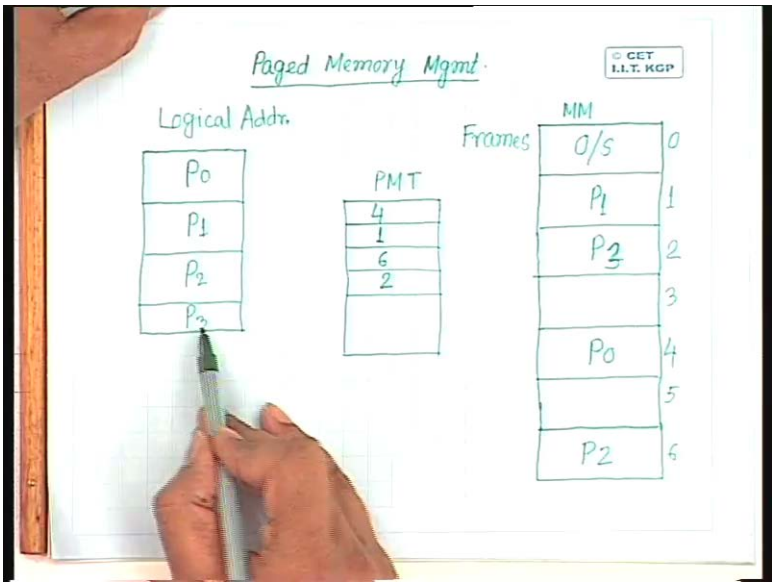
Digital Computer organization
Prof. P. K. Biswas
Department of Electronic & Electrical Communication Engineering
Indian Institute of Technology, Kharagpur
Lecture No. # 15
Memory organization –III

Till now we have discussed about two memory management techniques which are suitable for multi programming kind of application. So, one of the memory management that we have said is MFT or multi programming with fixed number of tasks in which case the main memory is divided into a number of partitions of fixed size. The partitions may be of same size or it may be of different size and other kind of memory management that we have seen is MVT or multi programming with variable number of tasks in which case there is no pre partitioning of the main memory but the partitions are created as and when there are needed of appropriate size. However the drawback in both the cases we have seen that if we have a job which requires a 100 kilo bytes of memory then we must have in case of MFT, a free partition of size at least 100 kilo byte. So, the size has to be more than or equal to 100 kilo bytes. Whereas in case of MVT; we should be able to create a free partition of at least 100 kilo bytes.

Now, if you have free partitions whose size is less than 100 kilo byte in that case that job cannot be loaded into the main memory. Whereas in case of MVT technique, we have seen that if we employ the memory compaction technique that is if we have a number of partitions, free partitions or free memory areas such that the sum of all those areas is greater than or equal to 100 kilo byte then we can go for memory compaction to create a free partition of 100 kilo bytes, when this job can be loaded. However this memory compaction is also a costly process because until once you start compaction, unless the compaction is over, the CPU cannot execute any other job. So, a better option will be that instead of partitioning only the main memory, if we partition the logical address space of the job also that is job is also divided into a number of partitions.

In such case it is possible that even if you don't have 100 kilo bytes of main memory available as a single partition but that may be say 10 kilo bytes of each partition, I have 10 such partitions so that total becomes 100 kilo byte. If the job is also divided into 10 partitions of 10 kilo bytes each in that case, these partitions of the job or the partitions of the logical address space can be loaded into different partitions in the main memory, even if though there are not contiguous and even in that case the job can be executed. So such a memory management technique is employed in what is called a paged memory management technique.

(Refer Slide Time: 00:03:44 min)



So, now we will take discuss about what is paged memory management. So, in case of paged memory management, the main memory is divided into a number of partitions where unlike in the previous case that is in case of MFT where the partitions, different partitions can be of different size. In case of paged memory management, we assume that every partition is of same size, so IP partition the main memory into a number of frames. Now these partitions are called frames where the frames will be of same size. Similarly the logical address space of an user program is also divided into partitions of same size and each of this partition of the logical address space is now called a page.

So the main memory, this is the main memory this is divided into number of frames and this is the logical address space of an user program. I say this is logical because the addressing generated for the user program is assuming that the first address of the user address space is zero. That is how the compiler generates the address. Only when the logical address space of an user program is loaded into main memory, then only concept of physical address comes. Because then I have to know that what is the address of the location in the main memory where a data or an instruction will reside but until and unless it is loaded into the main memory, the addresses that are generated those are all logical addresses.

So this logical address space is divided into number a partitions, each of the partitions will be of same size except the last one. The last one may not be of same size. Say for example if I generate that every partition, ever space size is of 100 kilo bytes and the total logical address space is 512 kilo bytes then I will have 5 pages of 100 kilo bytes each where the last page will consist of only 12 kilo bytes. Whereas in case of main memory; every frame will be of 100 kilo bytes. Now what I can do is if I name these pages as a page number 0, page number 1, page number 2, page number 3 and so on and the frame numbers are 0, 1, 2, 3, 4, 5. Let us have some more frames 6 like this. We have said that out of this some of the frames will be given to the operating system.

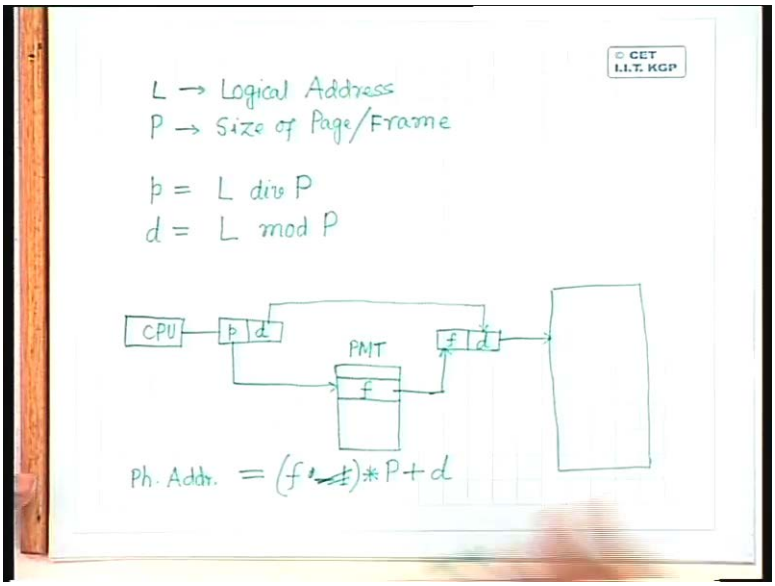
So, let us assume that operating system occupies frame number zero in the main memory. Now these pages of the logical address space can now be loaded in any frame in the main memory. It may so happen that say page number 0 is loaded into frame number 4 of the main memory. Page number 1 may be loaded into frame number 1 in the main memory. Page number 2 may be loaded in frame number 6 in the main memory and say page number 3 may be loaded in frame number 2 in the main memory. So now I can have a situation like this. When the CPU generates the logical address of an instruction or a data in the logical address space, the CPU does not know that in which of the frames this particular page is loaded. So I must have a translation mechanism which will translate the page number into frame number. So the translation is done with the help of a table which is called a page map table.

So what is the content of this page map table? The page map table will have an entry for every page of the logical address space. So since there are 4 pages in this logical address space, so there will be at least 4 entries in this page map table. So this is what is the page map table. So whenever the CPU generates the logical address corresponding to say page number zero, following the page number you have to come to the zeroth entry in the page map table or first entry in the page map table. This page map table will contain the frame number where this page number zero is loaded. So in this particular case, page number 0 is loaded into frame number 4. So this entry in the page map table will content the frame number 4. Then when the CPU generates an address, the address being in page number 1 then following this page number 1, this page map table will be searched and the next entry in this page map table will contain the frame number where page number 1 is loaded. So in this case page number 1 is loaded into frame number 1, so this entry will contain frame number 1.

Similarly the next entry will contain frame number 6 and the fourth entry will contain frame number 2. So whenever the CPU generates a logical address space, the logical address has to be converted into a two component address. The first component is the page number and the second component will be that what is the offset within that particular page. So using the page number, you come to the page map table. From the page map table you come to the corresponding frame number. And now we find that offset within the page is same as offset within the frame. so if I am looking for say fifth entry in page number 1 that will be the fifth entry in frame number 1 because page number 1 is loaded into frame number 1 or if I am looking for say tenth entry in page number 3 that will be same as tenth entry or tenth location in frame number 2 of the main memory.

So I have this kind of address translation, so firstly I have to decide that given the page map table for any logical address space, if it is broken into a two component address of page number and offset within the page, I can always calculate what is the physical address of that particular logical address by making use of this page map table. So now we have to decide that whenever the CPU generates an address, it does not generate the page number or offset within the page. The CPU generates a logical address say L.

(Refer Slide Time: 00:11:22 min)



Now we have to find out that from this L, how we can convert this logical address L into a two component address, page number and offset within page. So this L is the logical address which is generated by the CPU while execution of a particular process or a particular program. If we assume that capital P is the size of a page and as we said that the page size of the logical address space is same as frame size in the memory, so this will also be the size of a frame in the main memory. Now from this logical address and the page size, we can calculate what will be the page number and what will be the offset within the page. Simply by the operation, the page number if I put it as lower case p will be the logical address L, then integer division I put div for integer division by page size P. And offset within the page which is d will simply be the logical address. We have to use mod operation by the page size P.

Let us simply if you assume that the page size is say 10, 10 bytes and the CPU generates the logical address space of say 5. Then if we perform 5 div P, this being integer division the page number will be 0. And if you perform 5 mod P, the offset will be 5. So the logical address 5 is converted into a two component address with page number 0 and offset within page number 0 which is 5 only. Similarly if the logical address is say 17 then when you perform L div P that will give you page number 1. This being integer division L mod P, this is 17 so mod operation will give you 7. So a logic address 17 belongs to page number 1 and offset within the page is 7.

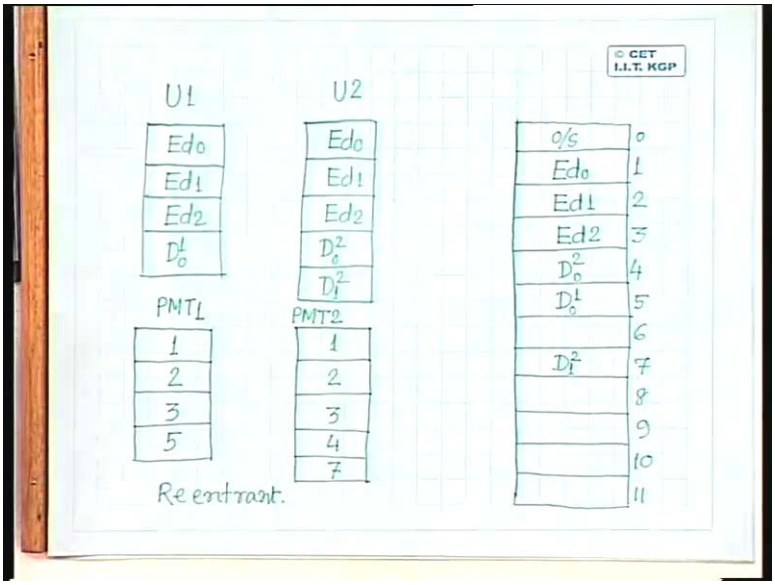
So in this case the CPU generates a logical address space. The logical address is divided into, is broken into two components. One is the page number p and offset within the page that is d. Using this page number, we have to go to the page map table. So the total scheme will be like this. Using page number, I come to a particular entry in a page map table. The page map table gives you the frame number where this page p is loaded. So I extract the frame number, combine this frame number f with the offset d.

So here I have frame number f, I make use of the same offset d because just now we have seen that offset within the page will remain same as offset within the form. These two together gives you the physical address within the main memory. And computation of the physical address is very simple because we know what is the page size or what is the frame size that is equal to p and f being the frame number, the physical address will be simply f minus 1 multiplied by the page size or the frame size P plus offset within the page or offset within the frame that is d. Starting frame from 0, that is why f minus 1.

So if I had an offset within the zeroth frame sorry, if you start from zero in that case minus 1 is not needed that is true. But if you use frame numbers from 1 then you need minus 1. So the physical address within the main memory will be f multiplied by P, assuming the first frame is the zeroth frame plus the offset within the page that is d. Now the advantage in this case is now it is no more needed that if I want to load a job of 100 kilo bytes, I have to have a single partition of 100 kilo bytes. So even if I have 10 partitions of 10 kilo bytes each and the partitions may be distributed throughout the main memory, even then I can load the job and execute it. There is another advantage with this paged memory management that is code sharing.

If we assume a situation that simultaneously two users are executing the same program. Suppose it is an editor program, two users want to edit their local files simultaneously by using the same editor say VI editor. Logically what should we have is because the user space or the user task for the two users will be different. So I should have two different copies of the editor program in the main memory. One copy for each of the users that means two copies of identical program should reside in the main memory in order to enable two users to execute edit program simultaneously, which can be avoided if we go for such a kind of paged memory management technique. In the sense that what we have to do is we have to manipulate the page map table according.

(Refer Slide Time: 00:18:12 min)



So suppose we have two user program, the user program 1 and user program 2. What is the logical address space of user program 1? Because the user program 1 is nothing but editing a file, file may be local to user program 1. The logical address space of user program 2 again will consist of two parts. One part will be the editor code and the other part will be local file of user program 2, of user two. So it may like this that the editor program consists of three pages. We will put that as edit program 0, edit page 1 and edit page 2. For user program 2 also, the editor program is broken into three pages edit page 0, edit page 1 and edit page 2. This may be D_0 of user program one that is the local file, the local data which is being edited by the edit program. For user program two may be the data consists of two pages one is D_0 2, other one is D_1 2.

The main memory map may be something like this. The first frame as before is given to the O/S and we have a number of other frames in the main memory. Size of every frame is same as the page size, so we have situation like this. These are frame numbers. Suppose the first frame is occupied by Ed_0, this is occupied by Ed_1, this is occupied by Ed page 2. Frame 5 may contain D_0 1, frame 4 may contain D_0 2 and frame 7 may contain D_1 2. So this is the zeroth data page of user program 2, this is the first data page of user program 2. Whereas the pages of the code edit that we want to be shared by both the user programs U1 and U2.

So in order to enable both U1 and U2 to share the same pages of the edit program or page map table for user program two should appear something like this. It has total of 4 pages. Page number 0 of user program 2 corresponds to Ed_0 and this entry in the page map table should contain the frame number where Ed_0 is loaded, so that is frame number 1. Next page of user program one corresponds to the edit code page Ed_1 and this entry in page map table should contain the frame number where Ed_1 is loaded. So we put here 2, similarly here we put 3. The fourth page of user program 1 is corresponding to the data page of user program 1 and that is a single page D_0 1. So this entry in the page map table will contain the frame number were D_0 1 is loaded that is frame number 5. So this is the page map table of user program 1 or PMT_1.

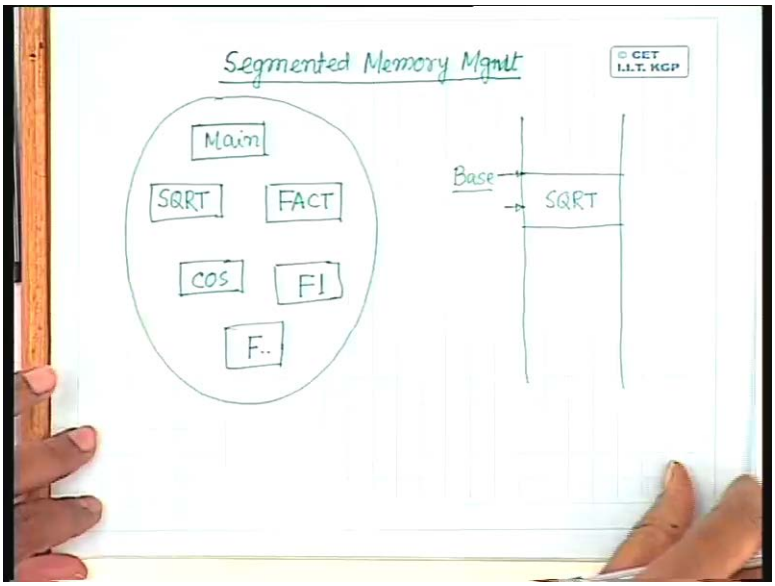
Similarly page map table of user program 2 that is PMT_2, there are 5 different pages, it will have 5 entries. First three entries should be identical because page number 0 of user program 1 is same as page number 0 of user program 2 and both of them are Ed_0. So first three entries in the page map table of user program 2 should be identical with the first three entries of the PMT_1. Next two entries will be different because these two are local for user program 2. So this entry should contain the frame number where D_0 2 is loaded and that is frame 4 and this should contain the frame number where D_1 2 is loaded and that is frame number 7.

So now we find that while execution of this edit program whether this is executed by user program 1 or it is executed by user program 2, whenever the logical address is generated the logical address will be broken into page number and frame number. If the page number is within 1 to 3, 0 to 2, 0 1 or 2 then whichever physical address user program 1 accesses, the same physical address will also be accessed by user program 2.

The moment it corresponds to data, the physical address accessed by user program 1 will be different form the physical address accessed by user program 2. So by making use of this paging concept, we can allow sharing of the same code by more than one process. of course the only restriction is the codes should not be self modifying that is while execution, the program should not modify itself and such a code is also called reentrant code because if while execution user program one modifies a part of the code because the same code is being shared by user program 2. User program 2 will get the modified code but may be the user program does not want the modified code it wants a raw put.

So if we allow more than one processes to share the same code, the code should not be self modifying or it should be reentrant code. So in this case you find that the logical address space is blindly broken into a number of pages, where every page is of same size except the last page or the last page can be of size which is less than the page size. In that case it will lead to some amount of internal fragmentation and because we have broken the logicals address space blindly, it does not take care of the modular structure of the program. Because these days we always encourage that whenever you write a program, the program should be structured program or modular program. but it does not take care of the modular structure and that may leads to difficultly in some cases and we will come to that, what kind of difficultly you can have when you discussed about what is called virtual memory management. So there is another kind of concept which is similar to this and that takes care of the logical structure of the program which is called segmentation and the corresponding memory management is termed as segmented memory management.

(Refer Slide Time: 00:27:24 min)



What is this segmented memory management? When you write a program, usually you break the entire task into a number of sub tasks. Every sub task you will try to implement either with the help of a function or with the help of a procedure.

Then you have a module in your program which is called a main function. From the main function you call all those different subprograms or procedures or different functions. So if you look at the logical address space, the structured logical address space of a program: it will consist of a main program and it will consist of a number of different functions. May be you have a function called square root, you can have a function called may be say factorial. You can have a trigonometric function may be finding out cosine of some angle, you can have some function say F_1 and there may be various such functions in your program.

Now all these functions taken together they form the logical address space of an user program. When you go for paged memory management, the paging does not take care of these functions. It simply considers that the inter address space of all these functions is a single logical address space. You simply break that logical address space into a number of pages or every page will be of same size. So in that case it is quite possible that a part of the main program, the main program will be divided into two pages. The first part will be content in one page; the second part along with the first part of SQRT will make another page. Then second part of SQRT may be the third page, last page of SQRT may be combined with another part of this function factorial to generate another page. So this logical structuring of this user user address space is totally broken.

In case of segmented memory management, what is tried is we try to maintain the logical address space of the structure of the logical address space that means which we will create a segment corresponding to the function main. We will create a segment corresponding to function SQRT; we will create a function corresponding to create a segment corresponding to this function factorial and so on. Now in this case you will find that different functions may be of different size. So the paging kind of technique which you have used in paged memory management technique may be paged memory management is no more applicable here because we cannot ensure that all the functions will be of same size.

If all the functions are of same size then my main memory management could have been same. I could have fitted this different function into different pages, if they are of same size but because in general the sizes are not same, the paging technique is not suitable for this. Rather the kind of technique which is suitable is what we have applied in case of MVT technique. That is you generate a partition of appropriate size whenever that is needed.

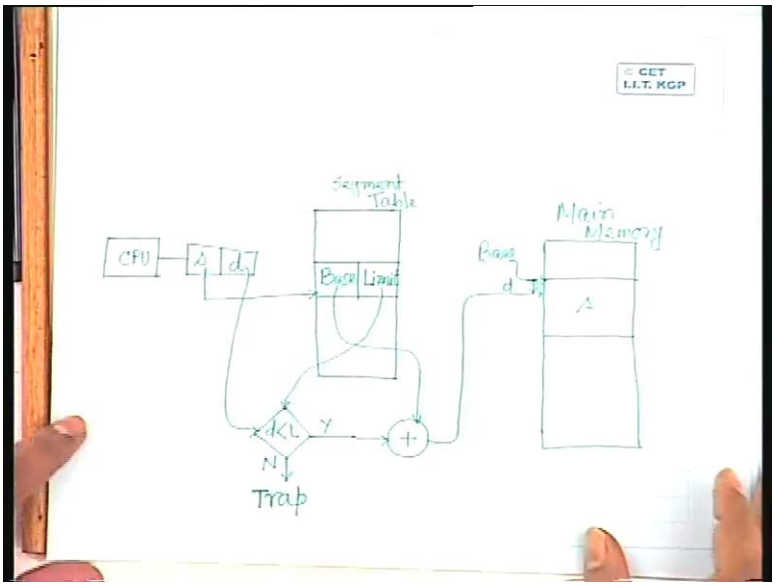
Now in this case all this partitions which are generated in the main memory, there are also segments and the segment sizes will be different depending upon whichever function has to be loaded into that segment. So if the main function takes the 10 kilo bytes of memory, in the main memory I will create a partition of 10 kilo bytes and load this function main in that main memory, in that particular partition. Again the other problem of MVT that we have been able to avoid that now it is no more necessary that if the entire address space is 100 kilo bytes, I have to have a single partition of 100 kilo bytes. But what is needed is that I should be able to create partitions of different sizes but those partitions should be able to accommodate in this functions.

If main function contains say 10 kilo bytes, I should be able to create a partition of 10 kilo bytes or this main function will be loaded. If SQRT is 40 kilo bytes I should be able to create a 40 kilo byte partition where this SQRT will be loaded. And now this partitions can be anywhere within this main memory unlike in case of MVT. So there lies the similarity that in case of paging, different pages could be distributed anywhere in any frame in the main memory.

Similarly in case of segmented memory management, different functions can be loaded in different segments distributed in the main memory. Only thing is the segment size should correspond to the logical size of different functions. And now again since we are distributing this functions in different segments throughout the main memory, I also have to have a mapping function as we have done in case of paged technique by making use of page map table. So now instead of a page map table, what I have to have is what is called as segment table because it is a segmentation that we are incorporating.

So whenever any function is loaded in the main memory, so if we have a main memory like this and I create a partition, suppose this partition accommodates the function SQRT. While compilation, after compilation I know what is the size of this function SQRT. So that will indicate that what is the limit or length of the function SQRT. So whenever a logical address is generated for this function SQRT, the logical address cannot exceed the maximum length of SQRT, so that puts a limit on this function. and the starting address of this segment where this function SQRT is loaded that we call as the base address. So once we know the base, what is the base address of a particular segment and what is the offset within that segment, if you add the offset to this base address, I can come to the physical address of the main memory which contains the desired instruction of the desired data which is requested by the CPU.

(Refer Slide Time: 00:34:47 min)



So the segment table for this segmented memory management will appear like this. The segment table as before will contain two entries, number of entries which is equal to the number of segments that we have within the program. Every entry in this segment table will contain two fields. So every entry will consist of two fields. One field will be the base address of the segment and the other entry will be limit of this of the segment; limit means what is maximum offset allowed within that segment that is nothing but the segment length.

Now the CPU generates an address. In this case the address will again be a two component address. I have the segment number s and offset within the segment that is d. using this segment number, I go to an entry in this segment table. The segment table gives me base address and the limit of the segment. The CPU generated the segment number and offset within the segment for a particular instruction or a particular data. My limitation is this offset should not be more than limit because limit is the length of that particular function. Whichever address, whichever offset is generated that must be less than this limit. So from this entry I take out these two components, the base address and the limit. Firstly, I compare d with this limit to check whether d is less than limit or not. If d is not less than limit that means the address generated by the CPU is wrong because this offset may lead to interfering with some other function or some other user program.

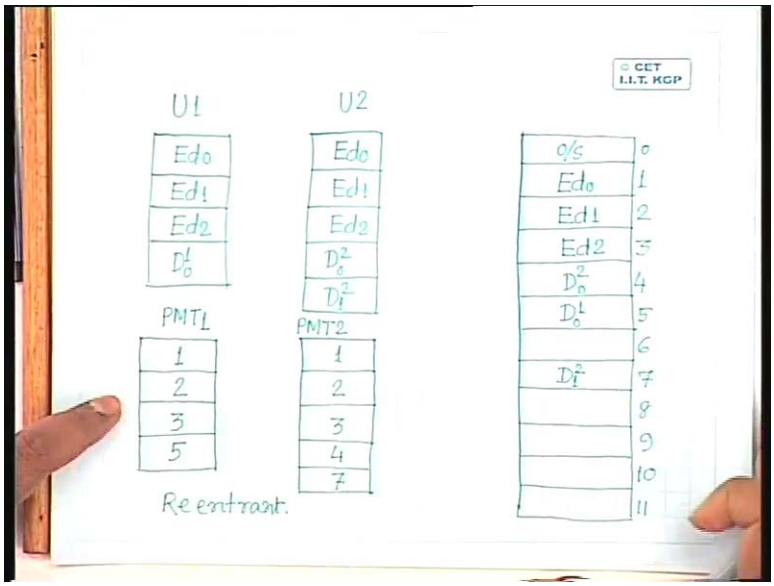
So if d is not less than limit, immediately there has to be an inter after trap following which the program execution has to be terminated. In case d is less than L, so this is no. If d is less than L, in that case what I have to do is I have to add this d with the base address because whenever you add base address with the offset that gives you the physical address within the main memory of the instruction or data that is required. So this is in the main memory which contains the corresponding segment and this is what is the offset d and this is the base address and this is the main memory. So we find that the concept is almost similar to paged memory management technique. The difference is that now the segments will be of different size unlike in case of paging technique where every partition is of same size.

The advantage is that this segmented memory management technique takes care of the modular structure of the program. Every function or every module will be a particular segment in this segmented memory management technique. However, there is one difficulty that is in case of paged memory management technique, we have seen that we can easily share codes by different programs. If we want to extend the same facility in case of paged memory management technique, in that case we have to ensure suppose the SQRT is a function which is shared by more than one user program. User program 1 wants to use the function SQRT square root, user program 2 also wants to use the same function SQRT which is square root.

In that case if we want to enable sharing of that function by more than one process, then what is needed is the segment number of SQRT in all the processes must be same. If the segment number is 5 for SQRT for user program 1, it also has to be 5 for user program 2 which is difficult to ensure.

However, if that is ensured then sharing of codes is also possible in case of this segmented memory management technique.

(Refer Slide Time: 00:40:10 min)



In other case also you have the same number but here what you have is if we assume, see what we have in this case, suppose instead of making this as page number 0 of user program 1, this is made as page number 5 of user program 1. In that case this frame number 1 will appear in the fifth entry of PMT_1. It will appear as fifth entry in the PMT_1 then also it can shared but here in case of segmented memory management, what you are doing is this segment number also is specified by the CPU. CPU itself is generating segment number and the offset within the segment. Here in case of paged memory management, what the CPU gives is the logical address L, from L you are calculating p.

So depending upon relative positioning of different codes or different functions within the logical address space, your p will be different. Is it okay? But it cannot be such that the same functions square root is being addressed as segment number 0 by CPU for use program 1 and it is being address as segment number 5 for user program 2. That is not possible because whenever it is square root, the segment number has to be specific then only the CPU can generate a unique segment number. In this of paging that is not needed, in case of paging depending upon relative positioning of different functions within the logical address space, its page number will be different and that page number is not given by the CPU. The page number is generated externally. So when you generate it externally, the relative positioning is all automatically taken care of. Is it okay?

So we find that we have two different techniques. The problem in case of segmented memory management apart from sharing is that because in this case I have to maintain different memory partitions of variable sizes. So main memory management may be slightly difficult than in case of paged memory management because in case of paged memory management, the partitioning is predefined but in this case the partitioning has to

be done depending upon which function is to be loaded. So managing partitions in the main memory may be difficult than in case of paged memory management technique.

However, it has got the advantage that it takes care of the modular structure of the program. So paged memory management is simpler in the sense that management of main memory is easy, segmented memory management takes care of the logical structure. So if we can combine the advantages of both then possibly we can have a better memory management which is called paged segmentation or paged segmented memory management and that we will do in the next class.