Digital Computer Organization

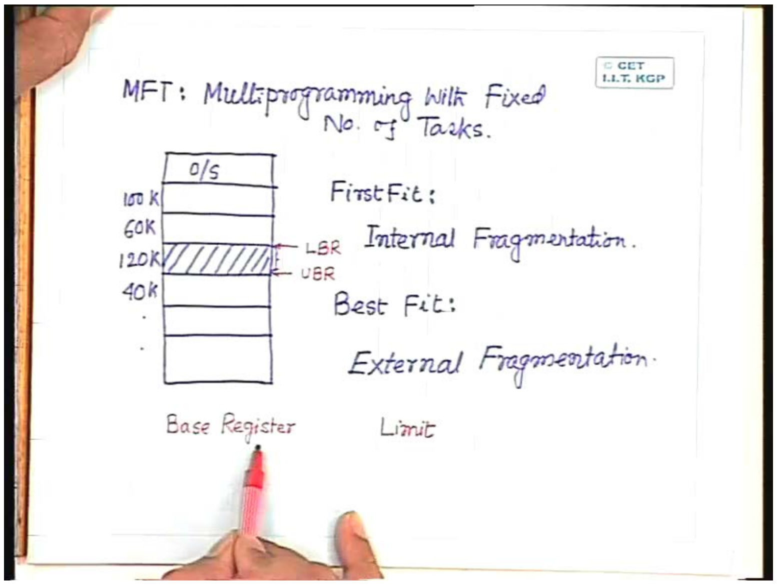
Prof. P. K Biswas
Department of Electronic & Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture No. #14

Memory Organization-II

I just forgot to mention one thing that in case of this MFT technique because we have a number of programs in the main memory residing simultaneously and any of them can be executed by the CPU any time. So we need protection of each user program against others.

(Refer Slide Time: 00:01:24 min)



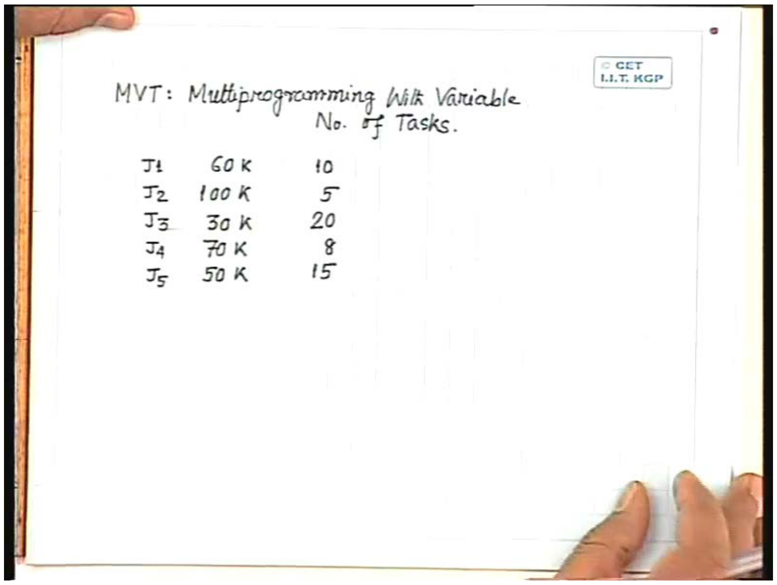
So for that what you have to do is for every partition, we have to have two registers with one of them we call as lower bound register and the other one is upper bound register. That means whenever the CPU generates the address, while execution of an user program for every address generated you have to check whether the address generated is greater than lower bound register content or not and less than or equal to upper bound register content or not. So only when it is greater than lower bound register and less than or equal to upper bound register that means within this region then the addressing is valid addressing.

The moment it goes below this, below the content of lower bound register or becomes more than the content of the upper bound register that immediately indicates that the program is trying to access some memory area which is not actually allocated to the process. So immediately some interrupt is to be generated like trap and this program has to be terminated. This gives protection of every user program against other user programs. So now we come to… So what modification can be done is instead of using this lower bound register and upper bound register as we have done before, we can have a base register. What the compiler can tell you is what is the length of the program that is what we will call as limit; that is the maximum logical address to be generated by that user program.

So what you can do is whenever a logical address is generated by the CPU because CPU is always generating the logical address. Whenever a logical address is generated by the CPU, while execution of some user program, the logical address can be compared with limit. So always the logical address has to be less than or equal to limit and obviously it has to be greater than zero. If it is within the limit then the content of base register can be added to the logical address to give you physical address in the memory. In that case the base register will be same as the lower bound register, upper bound register we can avoid.

So you have to add the logical address to the content of the base register to give you the physical address within the memory and because the logical address cannot be less than zero. So that ensures that will never go below this base register content. We are taking that it is not greater than the limit that ensures that we will not cross this upper bound register content. So that can be a variation on this, instead of using lower bound register and upper bound register, we can use the base register and a limit on the address space of that user program.

(Refer slide Time: 00:04:50 min)


MVT: Multiprogramming With Variable No. of Tasks.

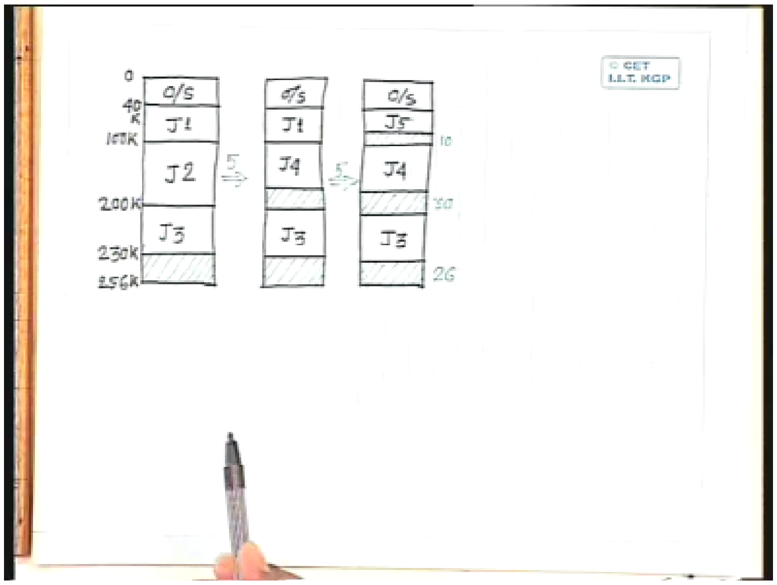
J1	60 K	10
J2	100 K	5
J3	30 K	20
J4	70 K	8
J5	50 K	15

So now let us come to what is called MVT or multiprogramming with variable number of tasks.
So you will find that in earlier case in MFT, partition size and the number of partitions were fixed, so the degree of multiprogramming was also fixed. If I have n number of partitions in the main memory, my degree of multiprogramming is always n, I cannot exceed that. But in this case since the partitions are variable and the partitions will be created as and when required. So I can have a variable number of tasks, so the degree of multiprogramming can also be variable in case of MVT.

So let us explain this MVT technique with an example. Let us assume that we have got say 6 jobs or let us start with 5 jobs. Say job number 1, job number 2, job number 3, job number 4 and job number 5. Suppose job number 1 the size requirement is 60 kilobytes, job number 2 the size requirement may be something like 100 kilobytes, job number 3 let us assume size requirement is 30 kilobytes, job number 4 let us assume that size requirement is 70 kilobytes and job number

5 the size requirement may be say 50 kilobytes. Time required for execution of job number 1 may be say 10 time units, for job number 2 it may be 5 time units, for job number 3 it may be say 20 time units, for job number 4 let us assume it is 8 time units, for job number 5 let us assume it is 15 time units. So this is the situation that we have. Let us assume that we have a memory something like this say 256 K memory.

(Refer Slide Time: 00:07:26 min)



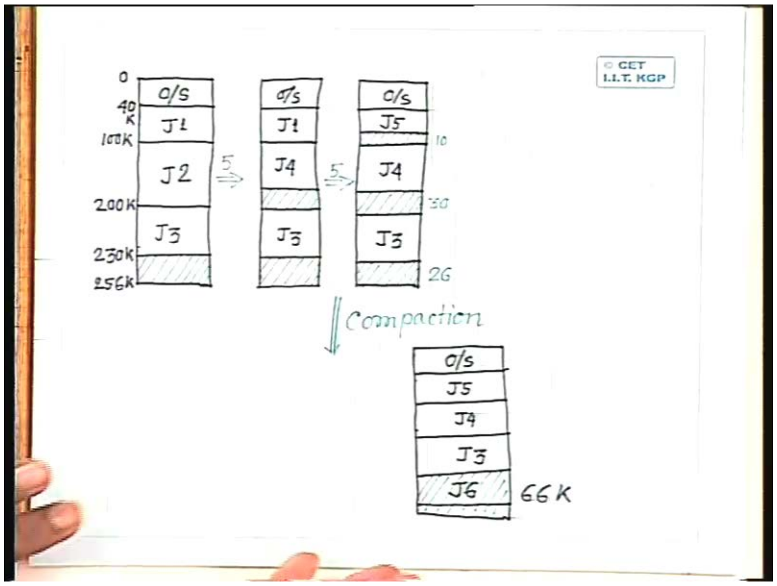
This is 256 K out of which let us assume that 40 kilobytes of memory is occupied by the operating system. So out of 256, 40 kilobytes of memory is used by the operating system. So remaining 216 kilobytes is the entire space. When we don’t have any job to be executed this entire memory is a single partition. The number of partitions will be created as and when required. The first job to be executed is job J1 whose size requirement is 60 kilobytes. So on getting this job, what you do is you divide this memory into two partitions. One partition of size 60 kilobytes and that is given to job J1. So here it is 100 K and the remaining 156 K is a single partition which is free.

Size requirement of job J2 is 100 kilobytes which is less than 156 kilobytes. So again this free partition is divided into two partitions. One partition of 100 kilobytes which is given to job J2 and so this is 100 kilobytes, this boundary is 200 kilobytes and we have remaining 56 kilobytes which is again a free partition. Come to job J3 whose size requirement is 30 kilobyte. So out of this 56 kilobyte, again you break this into two partitions, one of size 30 kilobytes which is given to job J3. So this becomes 230 K and I have a remaining free partition of size 26 K. I have two more jobs to be allocated job J4 and job J5. For j4 the size requirement is 70 kilobytes which is more than 26 kilobytes that we have free, for J5 it is 50 kilobytes which is again more than 26 kilobytes that is free. So I cannot allocate that partition to neither job J4 nor job J5. So these two jobs will remain in the waiting queue. So this is the situation now.

I have three jobs in three partitions and I have a remaining 26 kilobytes partitions which is free. So now we find that the execution time out of these three jobs which are there in the main memory J1, J2 and J3, execution time of job J2 is 5 time units which is the first job to be completed. So after 5 time units the situation will be like this. First 40 kilo byte is occupied by the operating system, next 60 kilo byte is occupied by job J1, next 100 kilo byte because job J2 is already complete will become free. Next 30 kilo byte will be occupied by job J3 and this is again another free partition of size 26 kilobytes. Now at this point, here I create a partition of 100 kilobytes which is free and the size required for job J4 is 70 kilobytes. So this 100 kilobytes is more than 70 kilo byte. So immediately what I can do is I break this partition, single partition into two partitions, one of size 70 kilobytes which is given to job J4 and the remaining partition of 30 kilobytes which is again less than the size requirement of job J5. Job J5 requires 50 kilobytes, I have created a free partition of 30 kilobytes. So job J5 cannot be allocated, so this also remains as a free partition and that happens after 5 time minutes. So I have so many partitions occupied and there are two partitions which are free.

The next job to complete execution is job J1 whose execution time was 10 time units. Out of these 5 time units is already over, so let us assume that all the jobs are getting CPU time. That does not change the actual scenario because all the jobs are executed in time multiplexed fashion. So may be this 5 will be 5 into 3, 15 times units because there are three jobs which are sharing the CPU. So let us assume that after 5 more time units, job J1 will complete execution. So after 5 more time units, job J1 will again complete and the situation will be something like this. Operating system partition will remain as it is, job J1 was occupying 60 kilobytes so that becomes free. Job J4 is occupying 70 kilobytes then 30 kilobytes is free, job J3 is occupying 30 kilobytes and then 56 kilobytes free.

(Refer Slide Time: 14:32)



So here I create a partition, I get a partition of size 60 kilobytes which is free whereas the size required by job J5 is 50 kilobytes. Now I can break this 60 kilo byte partition into two partitions like this. One of 50 kilobytes which can be allocated to job J5 and the remaining 10 kilobytes will remain free. So I have a situation like this that I have now 6 partitions, in addition to the partition given to the OS. Out of which three partitions are occupied by job J5, J4 and J3 and I have three partitions. This is a free partition of size 10 kilobytes, this is a free partition of size 30 kilobytes and this is a free partition of size 26 kilobytes. So you will find that the partitions are created as and when they are required. I don’t have any fixed partition unlike in the MFT case.

Now what is the advantage? You find that internal fragmentation is totally avoided because I create a partition of appropriate size, whenever that is needed and the partition can be created. So there is no question of internal fragmentation but there can be a case of external fragmentation. If in this list, I have another job say J6 whose size requirement may be 60 kilobytes. You will find that in this, there is no partition whose size is greater than or equal to 60 kilobytes. So this job J6 cannot be fitted into any of the partitions. Though the total free partition, the total free area is more than 60 kilobytes. This is 10 plus 30 that is 40 plus 26 that is 66. Though total free area is more than 60 kilobytes but still this 60 kilo byte job cannot be allocated because I don’t get a contiguous free area of 60 kilobytes. So all these partitions, free partitions becomes external fragmentation but there is no internal fragmentation.

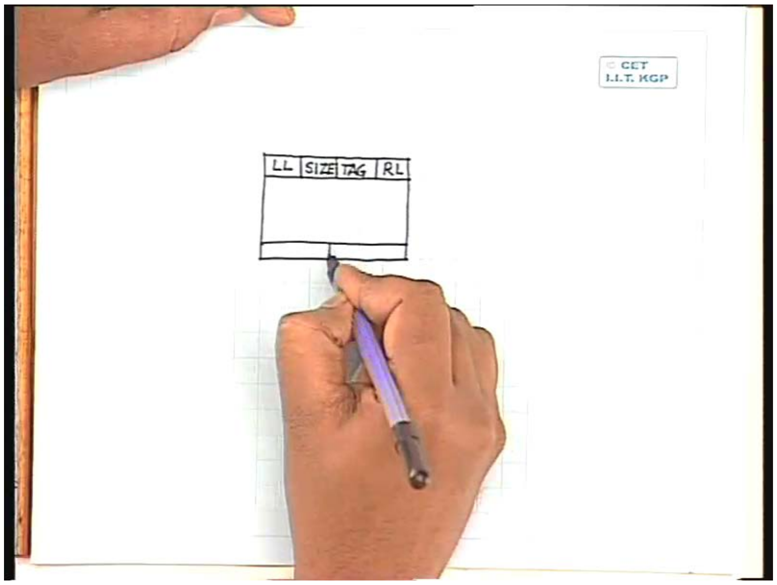
However this external fragmentation can also be minimized, if we make use of another technique which is called compaction. In compaction what we will do is we will try to move all the allocated partitions, occupied partitions to one end of the memory and all the free partitions to other end of the memory. So by doing that we can have a situation something like this. This situation of the memory can be compacted. So I have this OS then job J5 then you can have job J4 then we can have job J3 and then I have a free area of size 66 kilobytes which is greater than this 60 kilo byte size required by job J6.

So once I have this kind of situation, now this can be broken into two partitions. One a free partition of size 60 K and which can be allocated to job J6 and after that we will be left with a free partition of size 6 K. So this is a technique called compaction, memory compaction which can be used in case of MVT technique but this cannot be used in case of MFT technique because MFT by definition it is fixed partition. But what we need in this case? Because we are moving the job from one partition to another partition or shifting it from one side, one portion of the memory to another portion of the memory, so addresses which are used within that process within empty process have to be relocated at the address. I cannot have any fixed address and obviously we want to make use of the best register along with the address offset within that logical address space. The addresses are relocatable addresses so that can be easily done. But there is obviously some overhead that is whenever we have to move a block of memory from one location to another location then there is block transfer time which is involved and until and unless this block transfer is complete, the CPU cannot execute the process. But still this gives better memory utilization.

Now we have to see that if you want to employee this MVT technique, what are the information’s that we have to maintain? As in case of MFT technique, we have said that we have to have information for every partition regarding what is the starting location of that partition, what is the size of the partition and what is the status of the partition. Here also we have to have similar kind of information that is what are the partitions which are allocated and what are the partitions which are free. If there are partitions that are free then what is the starting location of that partition, what is the size of the partition? So for every partition I have to maintain those information whether the partition is allocated or the partition is free. This information can be maintained in the form of a table.

In case of MFT, the number of entries in the table are fixed. If I have say 50 partitions, I have 50 entries in the table. In case of MVT, the number of entries in the table is variable because the number of partitions is variable. So one method can be used that is the link list approach and it can be done in such a way that the link lists can be generated by making use of the free partitions only. I don't need any extra memory to maintain the link list information of the free partitions. So the link list can be implemented in this manner.

(Refer Slide Time: 00:21:16 min)



For every free partition we have in the memory, we can have certain fields. The fields can be something like this. We will have few fields at the beginning of the free partition and we will have few fields towards the end of the free partition. The fields can be something like this. I can have an R link field or right link field. All of you know what is link list. I can have what is called a left link field. I can have a size field. The size field will indicate what is the size of this free partition. I can have another field called a tag field which indicates whether the partition is free or partition is occupied towards the end. I will again have a tag field and I will have a field what is called an uplink field. This right link field points to the free partition which is next to this in the memory. Left link field will point to the free partition which is before this in the main memory. Uplink field will point to this free partition itself. Size field will contain the size value. Tag field will be equal to zero if the partition is free. It will be one, if the partition is allocated.

Now I mostly need the information about what are the partitions are free for allocation of a new job. This information is not needed for the partitions which are allocated to some process. So all the entries or the link list will contain the nodes corresponding to the free partitions only. It will not contain the nodes, the link list will not contain the nodes which are allocated. So now what you can do is whenever a job requests for some amount of memory, it may be a new job or it may be a job which is already allocated some partition and the job needs new memory by making use of malloc function. So whenever a job wants to have some more memory, the system has to check this link list. So there has to be a starting node following that starting node, for the starting node following this R link field you can move from one node to another node. Wherever free node you reach, you check what is the size field. If the size field is more than the amount of memory that is needed, this partition can be allocated. Whenever this partition is allocated, what you have to do is you have to bring this partition out of the free partition list. So accordingly you have to modify the R link field. Suppose this is the node which is being allocated, before this the free node that I have that is pointed by L link. I have to modify the R link field of that so that the R link field because this node is being allocated, so R link of this points to the next free node. So R link of the node before this that should now point to the next. So that link modification we have to do.

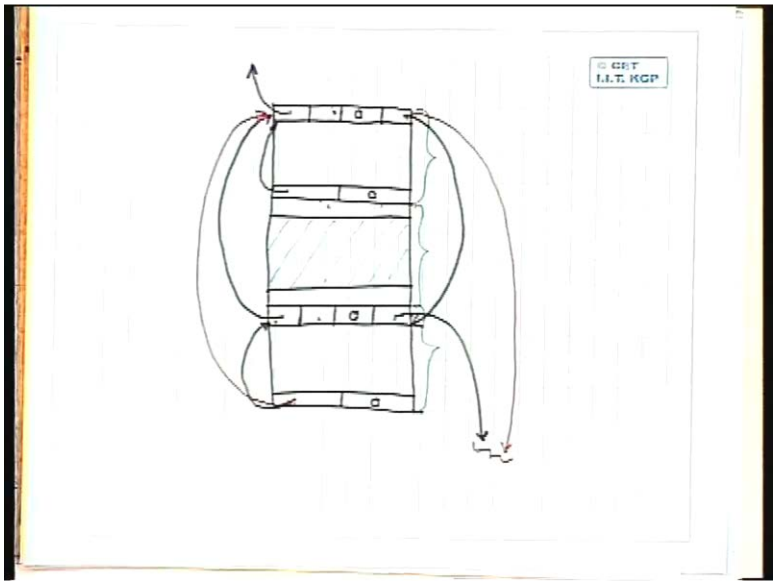
Similarly L link field of the next node has to point to the node before this. Tag field of this will be made equal to one, tag field here also will be made equal to one. I will come to that. So that is the modification we have to do. Now coming to the question of why do we need two tag fields. See in case of variable partition coming to this situation, suppose J3 now decides or J3 is terminated. That means the partition which is given to J3 is empty that becomes free, until if I don't take special precaution in that case even if this partition is free, I will have three free partitions in the memory. One of 30 kilobytes, this one of (how much is this?) 30 kilobytes, this one is of 26 kilobytes. So this will be maintained as 3 different free partitions 30 kilobytes, 30 kilobytes, 26 kilobytes. Though all of them can be clubbed together to give you an 86 kilobyte free partition and that is more helpful. That means whatever data structure we use for maintaining this free partition information, that data structure must facilitate merging of the adjacent free partitions. So to do that, this tag field as well as this uplink field they are very important.

Suppose the partition just after this which is already occupied by some processes is becoming free. When this becomes free, I have to check whether there is any partition adjacent to this which are also free. Now if they are also free then I can mark those partitions with this partition to get a larger free partition. Now how do I immediately check? Because this is having a pointer say some P. So whenever a process releases this partition, it tells you that point what is the pointer to that partition. So for as this partition is concerned, this does not have any information about this free partition or the free partition after this. There is where this tag field becomes important. Whenever I get P because this is the address of a memory location, I go to the location P minus 1, check what is the tag field at that location.

If at location P minus 1, I find the tag field equal to zero that means this partition is also free. Similarly this partition will also have a size field that is the size of the partition. I simply add the size value with this P, come to the first location of the next partition. This is P plus size, P plus size of P. Here also I can check, what is the tag field. If I find that tag field equal to zero, I know this partition is also free. So if I make P minus 1, I go to the tag field of the previous partition. If that partition is allocated, tag field will be equal to 1. If that partition is free, tag field will be equal to 0. If this is not there then what I have to do is I have to check this tag field. But how do you reach this tag field from value of P? That is not easy. I can check the tag field of the next partition by adding the size of P to P. I get this pointer but I cannot get this pointer easily. In that case what I have to do is I have to decrement P minus 1, access the uplink field. Using uplink field I have to go here. At this location I have to check the tag field, that is possible but it needs a number of operations. All these operations can be avoided by putting a tag field at the end.

So whenever a partition is made free, if I find that previous partition is free, let us assume that this is not free. If the previous partition is free, I have to merge these two partitions into a single partition. And for that what I have to do? I have to set uplink field here to this location. and what happens to R link and L link field? They remain as it is. They don't change but this R link field will change, if this was also a free partition. So by manipulating all these pointers, I can maintain the free partition list in the form of… Now you find that it is a doubly connected link list. I have pointers both in the forward direction as well as in the backward direction. So what I have is a doubly connected link list and this information is maintained by making use of the free areas only. I don't need extra memory to maintain this free area information. I don't need any table because in that case how many entries should be there in the table, that becomes a limitation. Because in case of a MVT technique, the number of partitions that had to be created is variable, so it is better to go for a link list implementation. and that link list can be implemented by making use of these nodes themselves. When merging, a new partition has been created so then it can obviously write whatever information is there. We can clearly write down to the fields in the intermediate… actually there are three partitions. There we at least three fields, I mean three different fields for each of them. So we can write down the information on black boards.

(Refer Slide Time: 00:31:16 min)



See for example suppose I have three partitions. I had three partitions something like this. This partition had a number of fields, this partition had a number of fields, and this partition also had a number of fields. So this is one partition, this is second partition and this is the third partition. Suppose this is the partition which was allocated and these two are the partitions which were free. So let me put it like this. This is an allocated partition and these two are free partitions. If that is the case then the situation before this partition is made free was something like this. The L link field of this was pointing to this partition because this one and this one are free, this one is allocated.

R link field of this is pointing to this partition, R link field of this one is pointing to the next free partition in the list. L link field of this is pointing to the previous free partition in the memory. It is having a size information which is size of this free partition. it is having a tag field which is equal to zero. Here also you have a tag field which is equal to zero and this is the uplink field which is pointing to this. Here also I have a tag field which is equal to zero, tag field which is equal to zero, uplink field is pointing to this. This is a partition which was allocated and now let us assume that this partition is becoming free. So when this partition becomes free then all these three partitions are to be merged together so that I get a single free partition.

Now if I wanted to merge them together, in that case what will happen? This partition was a free partition before, this partition was a free partition before and they were existing as two separate entries. Now all of them are being merged together, so I have to modify many of the pointers. What are the pointers that have to be modified? The first pointer that is to be modified is the R link field of this. This R link field does not have any more significance because these fields will not exist anymore. So I have to modify this R link field. and where it will point? It will point to the node where this R link field is pointing. So this R link field will move to the node where this is pointing. L link field I don’t have to change because there was no modification before this. What extra modification I have to do? That is the uplink field here. these uplink field is now pointing to this. Now all this three are being merged together, so this uplink field should point to this location. The other modification that I have to do is the size field.

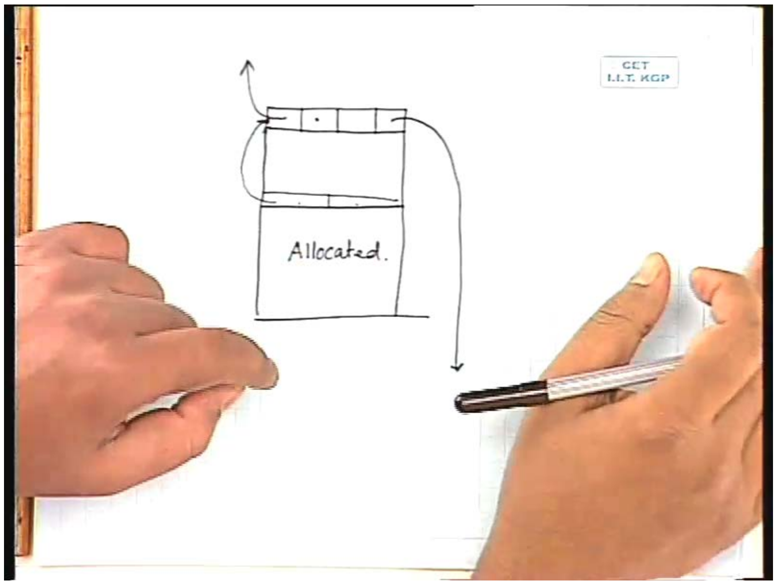
Now this size field was containing the size of only this partition. Now this size field will contain this size plus this size plus this size because the total size will be sum of all these size fields. These are the modifications that I need. I don’t need anything in between because these fields will not exist anymore. They will be overridden by the new data, whenever the partitions free. Is it clear? So now you find that as I was mentioning that in case of MVT technique, I don’t have theoretically any internal fragmentation though I can have external fragmentation. But in some cases it is profitable to introduce some internal fragmentation. trying to reduce the internal fragmentation to zero is not always logical. Simply because whenever I have to maintain a free partition, I have to maintain free partition in the form of this data structure. So in every free partition I have to have a link field, I have to have all link field, I have to have uplink field, I have to have two tag fields, I have to have a size field.

So for maintaining all these fields, a number of bytes are needed. Now you will find that if I have a partition of say 50 kilobytes out of which 49 kilobytes is being requested by a process. So I break that into two partitions, one of 49 kilobytes and a free partition of one kilobyte or may be the free time partition can be less than that. Then I have to find out whether maintaining such a free partition is profitable or not. because firstly, the one kilobyte free partition that I will maintain that may not be suitable or may not be use useful to any other process. And secondly to maintain that one kilobyte partition as a free partition, there is some overhead involved because I have to maintain so many fields.

So whether to go for a free partition of such a small size or you allocate that amount of the memory to the process who has requested for 39 kilobytes or 49 kilobytes and you maintain some small amount of internal fragmentation. So that is the tradeoff that you have to find. So that is why when the allocation is made in MVT technique, what is done is if you find that size of a partition is greater than the size required but the difference is very small. Then you don’t break that partition, you allocate the entire partition, let there be small amount of internal fragmentation. that is more profitable then breaking the partition into two partitions, creating a separate partition of a small area because whenever I introduce a new partition into the free list, the search time to search for a free partition is more. Because in case of a linked list; the search always has to be sequential. So, that is a tradeoff that you have to maintain and you have to see whether maintaining a small partition as a free partition is profitable or not. If it is profitable, you maintain it. If it is not profitable, you don't maintain it, just allocate to the process as requested and maintain a small amount of internal fragmentation.

So, though theoretically MVT technique can avoid internal fragmentation totally but practically it is more profitable to allow some internal fragmentation. What else? One more thing that if I find that a partition has to be broken into two partitions, one of them is to be allocated to the requesting process and the other partition is to be maintained as a free partition. Then how do you break the larger free partition into few partitions?

(Refer Slide Time: 00:39:03 min)



You will find that if you analyze this link list structure that suppose I have a free partition like this which is to be broken into two partitions. One of this will be allocated to the requesting process and the other one will be maintained as a free partition. then it is always better that you break it into two, the lower partition you allocate to the process, upper partition you maintain as free partition. Don't allocate from the top, you can allocate from the bottom. The reason being, if I allocate from the bottom the linked list remains as it is. Isn't it? This is a free partition which had a number of fields. L link field was pointing to previous free partition, R link field is pointing to next free partition.

Now, if I break it like this and this lower partition is allocated to the requesting process. These link lists, these pointers remain as it is. The L link field is not to be changed in any case because I am not doing any modifications on the upper side. R link field may be changed but if I follow this strategy that I will always allocate the partition from the bottom, the R link field is also not to be changed. It will also remain as it is because the next free partition has not changed. Only change you have to make is in the size field and here I have to create two more fields, one for tag field, and one for uplink field. This uplink field has to point to this. If I do it the reverse that is if

I allocate from the top, not from the bottom then all these pointers are to be created here that is more possible. So in case of MVT technique whenever you have to allocate a partition, you always allocate the, you have to break the partition. You allocate the bottom part of the partition and maintain the upper part as a free partition. So with this we stop our discussion today. Next day we will talk about other memory management techniques.