Operating System Fundamentals

Prof. Santanu Chattopadhyay
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

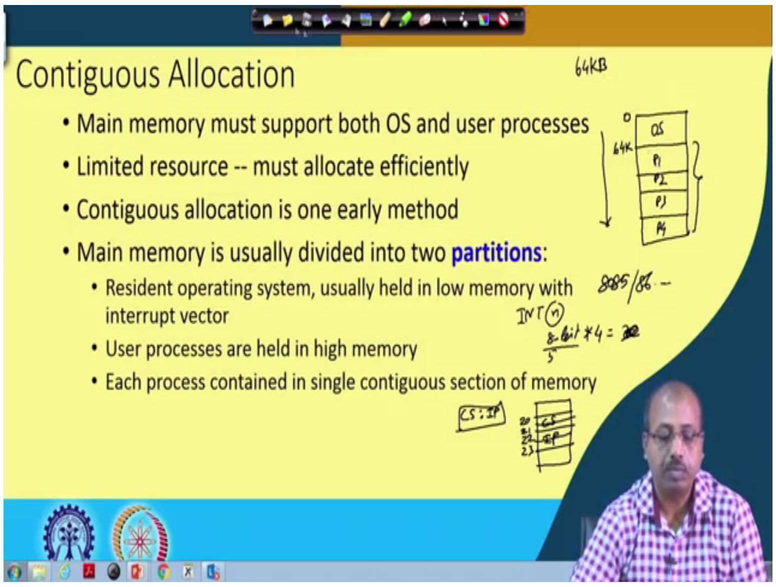
Lecture - 43
Memory Management (Contd.)

In our last class we have seen how a program can be compiled and then translated into machine code and finally, it is loaded into memory. And, we have seen that there are shared libraries that are attached with these modules so that the entire thing becomes executable. We have also seen a how a program may be loaded from different start address in the memory and based on that one base register is there so, which actually we loaded with the start address and that helps in program relocation.

And, also there was a limit register that was controlling how will how long is the program and then any address that is generated so, it should be between the start address and this limit register.

So that is the base register and base register plus limit register within that range; otherwise it can catch that there is the program is trying to access some address which is illegal which is not within the address space of this particular process. So after understanding those, so now, we will go to the detailed allocation policies for different schemes, for different memory management policies.

(Refer Slide Time: 01:32)



Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource -- must allocate efficiently
- Contiguous allocation is one early method
- Main memory is usually divided into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes are held in high memory
  - Each process contained in single contiguous section of memory

So to start with we will look into this contiguous allocation. So as the name suggests so here the entire memory space that is allocated to a process comes as a single chunk. So basically what we have is that a memory maybe if this is the full memory, so, out of that we find out a portion of memory which is free and this entire portion is maybe is allocated to process P 1. Similarly, some other part of memory so, that is again may be allocated to another process P 2. But, it is not that P 1 the memory allocated is distributed over number of regions in the program. So, it a number of regions in the memory.

So, it is not that if this is the full memory some part of P 1 is here, then another part of P 1 is here. So, it is not like that. So, it is coming together, entire thing comes together. So this is the simplest possible allocation that we can think about and this particular strategy is known as contiguous allocation. So, main memory must support both OS and user processes because operating system code itself will require some amount of memory. So what is done is that some portion of the memory is reserved for the operating system and rest of the part from there we allocate this user processes the memory locations.

So, typically if the is the memory so, initial portion of the memory is allocated for the operating system. So depending upon the size of the operating system if we assume that the entire operating system is loaded, and in this simplistic type of memory management so, entire operating system has to be loaded into the main memory and after that this entire region is available for user processes. So, here we can have different portions allocated to different user. So, this may be process P 1, this may be process P 2, P 3, P 4 like that.

So, limited resource; so must allocate efficiently because main memory size is not infinite. So, whatever be the big amount of memory which chip we put there still it has got a space limitation. So, compared to all the programs all the processes that the system can have, so, this space is limited. So, we have to allocate them efficiently. And, contiguous allocation is one of the primitive methods of this memory allocation. So, one the first method that was proposed so, that is the contiguous allocation.

So main memory is divided into two partitions one part is the resident operating system usually held in low memory with interrupt vector. So, basically we have got this memory address starting from 0 to suppose my memory with size 64K, so 64 kilobyte. So, that way up to 64 kilobytes so, this is the region allocated to memory. And, typically the lower order addresses, so address is increasing address is increasing in this order. So, this lower order addresses are allocated to operating system because of the reason that most of the processors that we have so, they have got some interrupt vector table. And this interrupt vector table is normally located at the initial part of the address space.

So, if you look into any processor for example, Intel 8085 processor or and say 86 processor so all of them, so they have got these memories this interrupt vectors located in the initial part of memory. So, typically they have the instruction like INT n and this n is a number which may be a maybe an 8-bit number. And this 8-bit number for example, in 8086 this 8-bit number is multiplied by 4, so, that gives a 32-bit that gives us 32 location 32-bit location. So, basically this 8-bit into 4, so, that is multiplied.

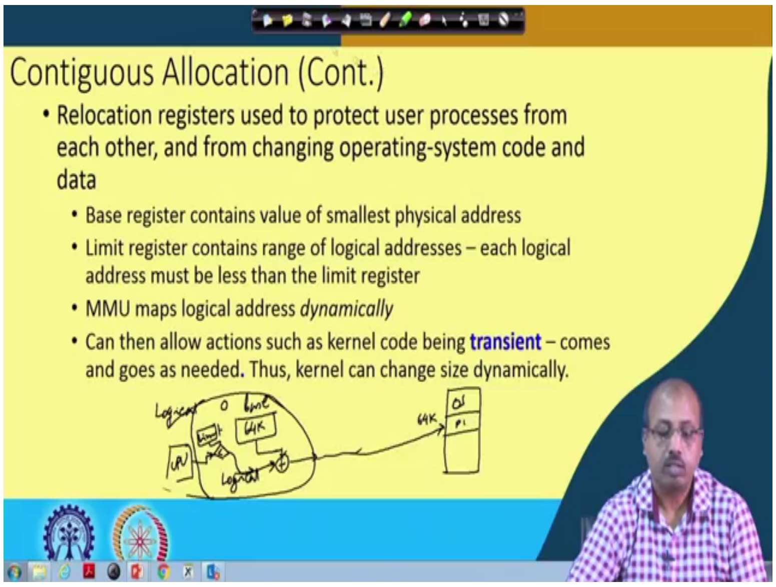
And then the out of that it there are two registers in fact, in 8086; one is called a code segment register another is called the instruction pointer registers. So, these two registers combined so that they actually give us the address of the memory location to be accessed. So, out of this 8 into 4; 8 into 4 that it is 8-bit location that we have. So, in that we have got four contiguous locations so, they are allocated. So, suppose this a bit number is say 5, so, the locations 20, 20, 22 and 23. So, they are the first two bytes they are used for storing the code segment register value and the next two bytes are used for storing the instruction pointer register value.

So that way this is a part of the interrupt vector table. So that way if I have got 256 possible interrupts because the value of n is equal to 8, so, 256 possible interrupts. So, 256 into 4 total 1 kilobyte space is required to hold the interrupt vector and as you can find as you can see here these values of the addresses that are generated are in the initial portion of the memory. Similarly, say 8085 also it is similar to the it has got ten on RST instruction and there also these interrupt addresses interrupt vector addresses are generated in a similar fashion and also in other processors that you look into.

So, every processor has got this documented address space for this interrupt vectors and this interrupt vectors so, they are generally located at the beginning of the memory space. So, that is why they are made part of the operating system and this operating system may be using this interrupt for some specific purpose. So, that way they actually try to put this interrupt vectors as part of the operating system. Though it may be programmable by the user in some systems maybe some of the interrupts are available to the user, but many of them are captured by the OS itself. So operate so, this operating system is put at the beginning part of the memory and that also holds the interrupt vector.

Then user processes are held in high memory. So, after this OS is over so, remaining part of the memory so, this is given to the user processes. And each process contained in single contiguous section of memory. So as I was telling that it is not that the code of P 1 is distributed over a number of regions in the memory. So, it is held contigually. So, that is the contiguous allocation policy.

(Refer Slide Time: 08:22)



Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** – comes and goes as needed. Thus, kernel can change size dynamically.

Now, in this policy relocation registers are used to protect user processes from each other and from changing operating system code and data. So, what we have, so we have a previously you have seen that whatever address is generated from the CPU, the logical address. So, that logical address assumes that the process starts from address 0. Now, if we have got this as the memory and say this is the resident OS and my process P 1 is starting from say the location 64 K.

So what is done is that this base register is loaded with the value 64 K and then so, this is the base register and whatever address is generated from the CPU. So, the CPU generated address so this is basically added with this base register value and so, this is the logical address and then this gives me the physical address. So, this is the physical address. Now this physical address is checked whether it is within the limit or not and then finally, it goes to the code of the P 1. So, if this is beyond this region so, if you so that way it ensured that P 1 when P 1 is running so, this is if this base register is loaded with proper value. So it will access the portion of the memory which is available for P 1.

So base register contains the value of smallest physical address and there is a limit register that checks the range of logical address. So, you can have a limit register here, can have a limit register here that is loaded with the size of the process and you can have a check whether this limit register whether the address that is generated is that is generated is less than this limit or not.

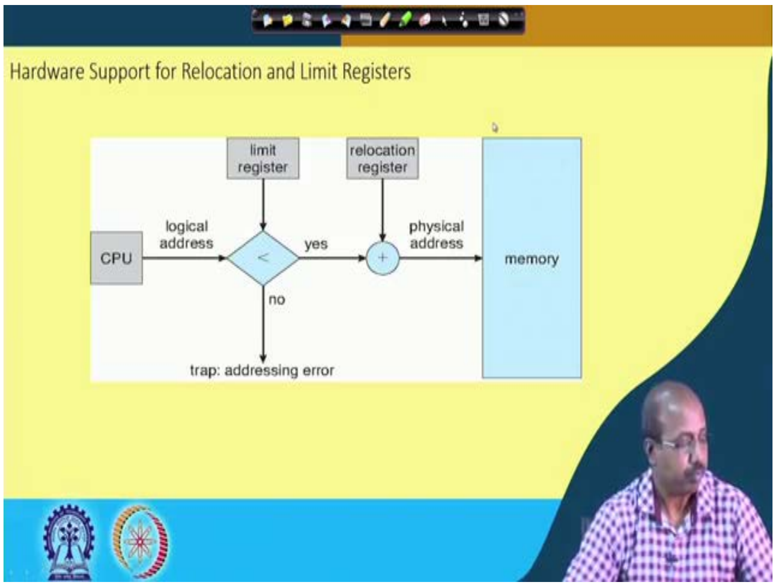
So if it is less then only it is allowed to go like this. So, you can say that this part is going like this. So, we have the if the limit if the number address generated is less than the limit then only it comes to the base register otherwise this is an addressing error. So, this way base register will be containing the smallest physical address.

And limit register it contains range of logical addresses each logical address must be less than the limit register. MMU, the Memory Management Unit it maps logical address dynamically. As we discussed previously that this entire thing this entire operation is part of the MMU.

So, this is sorry, not this CPU part so, if we just forget the CPU part. So, this address this limit register and then this checking and we are adding with base register. So, all this part so this is done by the memory management unit and that is that happens dynamically. So, the presence of memory management unit will help us in making this memory access fast.

Can then allow actions, such as kernel code being transients – comes and goes as needed. Thus kernel can change size dynamically. So, some of the systems what will happen is that OS size varies. So, sometimes the it has the flexibility like some part of the code may or may not be needed at some point at some point of time. So, that way this transient part may be there and that way this OS portion may shrink or expand a bit. So, that facility should be provided. Some mechanism has to be provided, so that this kernel code can be made transient.

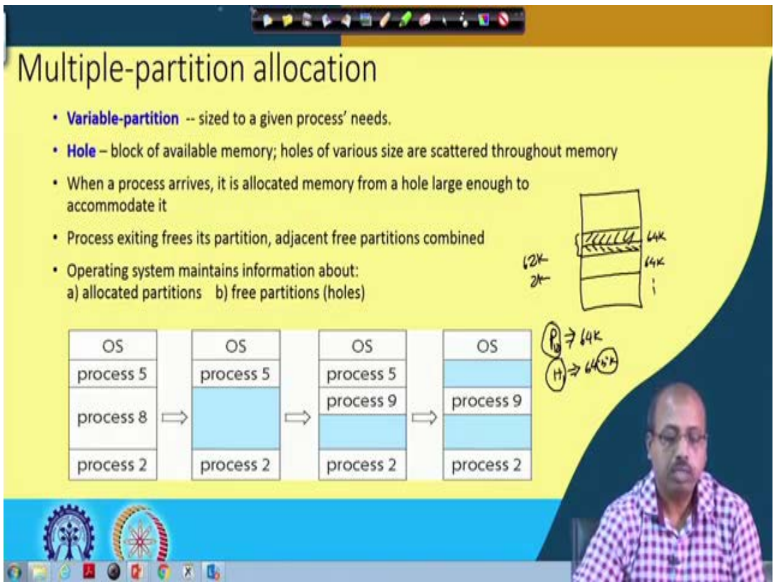
(Refer Slide Time: 12:09)


Hardware Support for Relocation and Limit Registers

So, next; so, next we will see that this diagram that is the CPU generates the logical address. So, logical address is checked with the limit register value and if this if the logical address is less than the limit register value then it is fine. Otherwise, it generates an a trap which is an addressing error and this addressing error that will be; that will be informed with the operating system and in turn the operating system will take care of the situation that there is an error. So, the process has to be restricted, either maybe process may be told to restart or something like that, but the process has developed some addressing error.

Then if the value is less than the limit registered in that case it goes to the goes to this adder, where this relocation register value or the base register value is added with this logical address and that generates a physical address for the memory. So, this is the hardware support that is provided for relocation and limit register. So, this all these things they are done by the memory management unit.

(Refer Slide Time: 13:14)



Multiple-partition allocation

- **Variable-partition** -- sized to a given process' needs.
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions b) free partitions (holes)

Now, there can be multiple partition allocation. So far whatever we have discussed so, there we have got only one partition allocated for a process and then these partitions so, they are also known as fixed partition. Like the scheme that we discussed like if this is the memory, a part of it is a dedicated for the operating system. Now, in the simplest possible strategy what we can do? We can just divide this memory into equal sized partitions. Each partition is of same length. So, any process coming so, it is allocated to one of this partition.

So, initially all the partitions are free and then the process P 1 came, so it is given the first partition. Then, the process P 2 came say. So, P 2 was given the second partition. So, then P 3 came, P 3 is given the third partition. Now, suppose P 1 finishes, so, this part becomes free again.

And then suppose another process P 4 comes so, P 4 may be allocated space here or it may be allocated space here. So wherever we have got a free partition the process may be loaded there. So, there the operating system designer or the system administrator has to have some idea about the sizes of these processes and based on that the partition size should be such that it is larger than the largest possible program that can run in this system.

So it is a bit difficult to predict of course, but initial memory management systems so, they did that so they are the partitions were kept sufficiently high so that the most of the user programs will fit. And so, if some user programs do not fit then the user may be asked to redesign the code or reduce the size of this data segment or something like that. Somehow that has to be arranged.

And many strategies were developed like something called overlay. Like say it is not that all procedures of a program is needed at every time. So, if we find that two procedures are needed in a mutually exclusive fashion that is when procedure 1 is being used, procedure 2 is never going to be used during that time. So, in that case we can allocate same space for procedure 1 and procedure 2. So, that particular strategy is known as the overlay. So, these were done to say have programs run which are larger than the partition. So, overly was one mechanism for doing that.

However, another possible way of handling this variable sized processes; like processes did not have a fixed highest memory highest memory requirement. So, we come to the notion of variable partition. So, sized to a given processes need. So, initially you can think that this we have got a situation where this hole memory we have got this operating system part and this entire remaining part is free.

Now, when a process comes say process P 1 comes P 1 declares how much space is it needs. Suppose, it tells that I need 10 kilobyte. So, it is given memory here. So, 10 kilobytes space is given to process P 1. So, we have got 40 kilobyte so, if this total was 50 kilobyte, now, this 40 kilobytes is left. So, next the process next another process comes so, it is allocated space in this 40 kilobyte region.

So, the point is like this; so, at any point of time if you take a snapshot of the system it may happen that the we have got process three processes 2, 5 and 8 currently running. And then this process 8 finishes after sometimes. As a result, so, this part of the memory becomes free. Now, another process 9 has arrived and when process 9 has arrived so, it has to be given some space ok. So, from this hole that is there, this is the available memory space. So, from here the depending upon the size of process 9, so, it is given a chunk of memory. So, this is the thing that is given. So, this part of the memory is now remaining.

Now, after sometimes our process 5 also finishes. So, this hole also the this memory space also becomes available. So, now, we have got two holes in my memory. So, this is one hole and this is another hole. So, holes need not be of same size, they are of variable sizes and then naturally the next request coming so, we have to see like which hole it fits into.

So, this variable partition at a sized to a given process needs and then a hole is a block of available memory. Holes are various sizes are scattered throughout the memory. So, after some time if you take a snapshot of the memory when the system is running, you will find that there are many such holes that have been created in the entire memory of space. When a process arrives it is allocated memory from a hole large enough to accommodate it. So, that is the policy. So, system has to keep track of the holes that are there in the system memory space and then when a new process comes, so, it is it searches for the appropriate sized hole and that way it is allocated the space from that hole.

And process exiting freeze its partition; when a process finishes, so, it freeze the partition. So, if we find that suppose after sometimes this process 9 leaves, so as a result these three holes that there that so, they can be combined into a big hole. So, adjacent free partitions are combined when a when a process leaves. Operating system maintains information about allocated partitions and free partitions, that is hole. So, it has to keep track of all these information.

So, fine; so, what is happening is that unlike a fixed partition so, where if the program is slightly larger than the partition size so, we could not accommodate it in the memory. So, that way, though that program could not be run in that memory management system. So, but with this variable partition size so, these partition size can vary. So, if we can find a hole of sufficient size then the program can be loaded into that particular hole and it can be done. So, we do not have a fixed partition.

So, fixed partitioning has got another difficulty also because suppose this fixed partition the. So, we have created a partitions and the each partition is of size say 64 K. So, this is a 64 K, this is 64 K like that. So, we have created each partition of size 64 K. Now, it is very much unlikely that the processes will require exactly 64 K. So, if it requires more than 64 K, then in a fixed partition scheme we have said that we cannot accommodate the process.

But, if the process requires for example, say 62 K; then it is allocated in this space so, basically uses so this part of the memory. So, remaining 2 kilobyte of space so, that is actually wasted. And this part, so, you cannot use for any other process also because this entire partition is allocated to the process so, that part remains unutilized. So, this particular problem is known as the internal fragmentation problem. So, but the difficulty is that you cannot utilize that internally fragmented memory space.

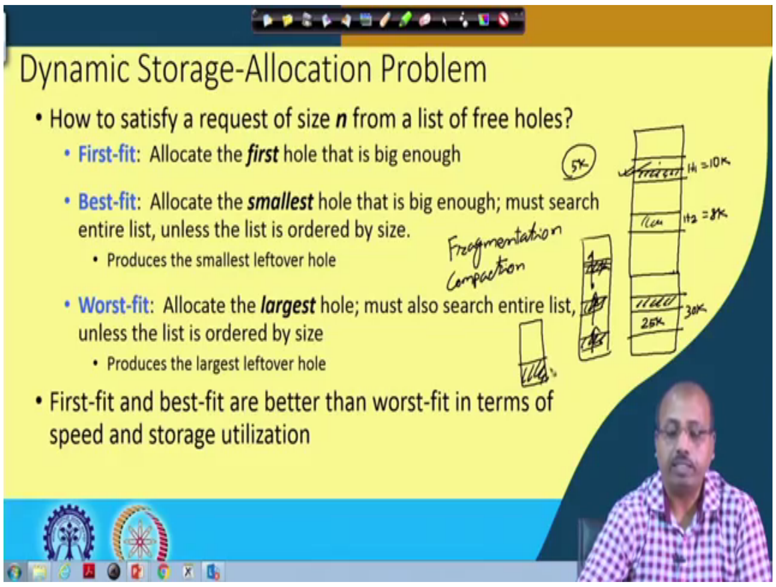
Unlike that in multiple partitions scheme with or variable partition scheme this internal fragmentation problem will be reduced to a significant extent because of this partitions themselves can be a variable size. Of course, at some point of time you have to accommodate that internal fragmentation because of the reason that suppose I have got a request for a new process say P 10 has arrived and it is requirement is say 64 K, fine.

Now, the part the holes that we have suppose we have got one hole H 1 whose size is say 64.5 K and we have decided that this process ten will be accommodate in this hole H 1. Now, you see this 0.5 K of space that is extra.

So, you can remember this 0.5 K space as a hole separately, but the overhead of remembering that space in the table maybe more because then whatever is. So, it is very much unlikely that a process will come whose requirement will be less or equal 0.5 kilobytes of memory space and then this so, instead of remembering that hole separately so, we can give this entire hole H 1 to process P 10.

So, that is that part of 0.5 K space so, that will be wasted. But, anyway, we are we can we cannot take care of that. So, that way the internal fragmentation remains there, but the extent of this internal fragmentation that reduces. So, we have got this variable partitioning scheme and with that we can accommodate the processes like this.

(Refer Slide Time: 22:37)



Dynamic Storage-Allocation Problem

- How to satisfy a request of size *n* from a list of free holes?
  - First-fit: Allocate the ***first*** hole that is big enough
  - Best-fit: Allocate the ***smallest*** hole that is big enough; must search entire list, unless the list is ordered by size.
    - Produces the smallest leftover hole
  - Worst-fit: Allocate the ***largest*** hole; must also search entire list, unless the list is ordered by size
    - Produces the largest leftover hole
- First-fit and best-fit are better than worst-fit in terms of speed and storage utilization

Now, the next question that we would like to answer is how to satisfy a request of size n from a list of free holes. So, we have got in the entire memory space so, we have got these type of holes created at any point of time if we look into so, maybe we have got a few holes here. So, this is one hole H 1; so, this is another hole H 2, like that suppose we have got another hole H 3. So, it is there. These holes are there. So, any process coming so, we have to accommodate it in one of these holes ok.

So, the request has come for a may process of size n. Now, the policies that we can follow, so, these are the three policies: first-fit, best-fit and worst-fit. So, first-fit it says that you scan from the beginning of the memory space of the memory and whichever hole comes first of sufficient size, you allocate it to the you allocate the process to that hole. Suppose, the process P 5 has arrived and this hole is of size say 10 K this hole is of size say 20 K and this hole is of size say 25 K.

Now, if you are doing a first-fit type of approach then it will start scanning from the beginning. So, first it gets this particular hole H 1 and H 1 is not sufficient. So, P 5 is of 15K, so, H 1 is not sufficient. So, it comes to H 2 and finds that the in this hole I can fit it ok. So, this P 5 will be fit into this hole H 2; so, the first-fit. So, it may so happen that this process P 5 maybe the situation is like this. So, we have got these holes in the memory. So, we have got a hole H 1 of size say 10 K, we have got another hole H 2 of size say 8 K.

Now, if there is a request for a process of size 5 K; so, first-fit algorithm so, it will find this hole H 1 and it will allocate the process to this hole. As a result it will create another hole of size 5 K here. So, that hole will remain. So, it will be allocating space in this part. So, the whichever hole comes first from the beginning of the memory that fits the particular process so, that is utilized for allocating space for the process.

Then we have got the best-fit policy. The best-fit is really the thing that we would intuitively like to do. So, whichever hole fits best, so we want to allocate that in a that particular hole for that process. So, it says that allocate the smallest hole that is big enough. So, in this particular case, so, you can see that the request was for 5 K. So, it is it will find that the H 2 is better because H 2 if I allocate, so, this is the smallest hole of size greater or equal 5 K. So, it is sufficient for the entire process. So, that way it is it will be chosen. Unless the list is ordered by size; so, this is so, whichever list whichever space fits first so, that best so that will be allocated in the best-fit policy.

So, the advantage of best-fit is that it produces a smallest leftover hole. So, definitely compared if you compare between first fit and best-fit it is very much likely that best-fit will have minimum amount of wastage of space because of the reason that it is trying to leave as little space unutilized as possible.

Then a counter logic for this is the worst-fit. So, worst-fit tells that instead of giving space like that so, what we do suppose we have got another big hole here which is of size say 30 K, then it says that you allocate this 5 K from this 30 K space. So, you give it here the reason or the logic for doing this is like this that if I give it from this space, the remaining space will be 25 K and that is possibly a big enough hole to accommodate some process later. Whereas, if I allocate it using best-fit or first fit policy the if the amount of space left particularly in the best-fit policy the amount of space left in the hole will be small and it is very much unlikely that the new hole that is remaining there so, that will be able to accommodate some other process.

So, that way worst-fit we can advocate for worst-fit looking into this idea, but at the same time worst-fit is definitely if we do simulations, then it has been seen that worst-fit really performs poor compared to this first fit and best fit. Though this first fit and best-fit apparently appears that the best-fit will do better, but there is no clear cut supremacy of either of them. So, but of course, you can always create some case or some example in which in that first fit will be better than best-fit and worst-fit. And also you can create other example cases where one of these policies may be shown to be better than others the best-fit or even worst-fit can be shown to be better than others. So, better than others in the sense that no so, that particular policy is able to allocate all the memory request for the processes, but the others could not.

So, this way you can have this partition allocation for this processes and as I was telling that this can solve the problem of this can solve the problem that I can have this size limitation is not there, but what happens is that it creates the problem of fragmentation. So, as we can understand that after some time if you take a snapshot of the memory so, it may so happen that this entire memory space so, it is it has many holes created in it. So, if you take a snapshot so, you may find that there are many holes that are remaining in the system. So, these are all the holes that are there.

Now, when a new request for a process comes so, maybe we find that none of this holes are sufficient they are sufficiently big to accommodate that process. But, if you take all these spaces together then we can create a enough space for accommodating the new process. So, what we need to do in that case is, we have to do something called compaction, we have to do compaction in which we push all these holes to one end of the memory.

So, you can if we can copy these processes up, similarly these processes up, then what will happen is that this entire hole will come to the bottom of the memory this entire hole comes to the bottom and there we can accommodate the new process. So, that compaction will be necessary. But, the compaction as you can understand is a very costly operation and that way it is difficult to do. So, until and unless we are failing to accommodate any many of the new process requests so, we should not try this. But, definitely we have to try this at some point of time otherwise memory will become fully fragmented.

So, we will continue with this in the next class.