

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

**Lecture - 44**  
**Memory Management (Contd.)**

In our last class, we have seen that this dynamic portioning policy, that we discussed. So, that has the potential to create large number of small sized holes in the memory space.

(Refer Slide Time: 00:38)

**Fragmentation**

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous and therefore cannot be used.
  - First fit analysis reveals that given  $N$  allocated blocks, another  $0.5 N$  blocks will be lost to fragmentation
    - 1/3 of memory may be unusable -> **50-percent rule**
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory.
  - Can happen if there is hole of size 15,000 bytes and a process needs 14,900 bytes; Keeping a hole of size 100 bytes is not worth the effort so the process is allocated 15,000 bytes.
  - The size difference of 100 bytes is memory internal to a partition, but not being used

The slide includes two diagrams: the top one shows a memory block with a 10k hole and a 5k process, with a 12k request  $P_i$  that cannot fit; the bottom one shows a 15k hole and a 14.9k process, with a 100-byte internal waste. A video inset shows the professor speaking.

And this particular problem is known as the problem of fragmentation. And this fragmentation it can be classified into two categories; one is called external fragmentation, another is called internal fragmentation. So, external fragmentation tells that the total memory space exists to satisfy a request, but it is not contiguous and therefore, cannot be used. So, this is the thing that I was talking about in the last class. So, the total memory space that is available in terms of these holes.

So, they are sufficient to put this hole may be of size say 10k, this hole may be of size say 5k. Now, if you have got a new process request  $P_i$ , that has arrived and it asked for 12k. Then none of the holes are sufficient to hold it. So, what we need to do? Is to do a compaction of this memory space and then by copying these processes to one end and moving this holes to the bottom of the memory. So, as a result it creates a hole of size

15k and then we can accommodate this 12k process here. And as I was telling that this is a costly operation because all the process codes are to be pushed, all the relocation registers were the process have to be updated. So, that the processes they can continue without any difficulty.

So it has been seen that the first fit analysis reveals that given  $N$  allocated blocks. Another  $0.5 N$  blocks will be lost to fragmentation. So, this is called one third of memory may be unusable, so that is 50 percent rule. So, it is; so  $N$  if you have got  $N$  allocated blocks, another  $0.5 N$  half of that is  $N$  by 2 blocks will be wasted.

So, due to this fragmentation, this is known as the one third of memory unusable, so that is 50 percent rule. So, this is the external fragmentation. So, external fragmentation means, the holes are there, but the holes are not big enough to accommodate the newly requesting process. But the sum of those holes, is good enough for the process. Then we have got another start type of fragmentation, which is called internal fragmentation.

So, the problem that we have is the allocated memory may be slightly larger than the requested memory. So, as I was telling that suppose this partition is of size say 10k, this hole is of size 10k and a process  $P_i$ , it requires a 9k of space. So if it requires 9k of space, so I will allocate it in this region in within this hole and this 1k that remains at the bottom, so that becomes unusable.

Now, so it is this is an example here. So, suppose we have got a hole of size 15000 bytes and a process needs 14900 bytes. So, keeping a hole of size 100 bytes is not worth the effort, so the process is allocated 15000 bytes. So, why do we say it is not worth the effort? Because the operating system needs to maintain a table containing all these hole information. Otherwise, when a new request comes how will it check, like when are the; which are the holes available and what are their sizes. So, it has to maintain a table.

(Refer Slide Time: 03:59)

**Fragmentation**

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous and therefore cannot be used.
  - First fit analysis reveals that given  $N$  allocated blocks, another  $0.5N$  blocks will be lost to fragmentation
    - 1/3 of memory may be unusable -> 50-percent rule
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory.
  - Can happen if there is hole of size 15,000 bytes and a process needs 14,900 bytes; Keeping a hole of size 100 bytes is not worth the effort so the process is allocated 15,000 bytes.
  - The size difference of 100 bytes is memory internal to a partition, but not being used

Start	Size
1000	15k
2000	5k

The slide also features a video inset of a man in a checkered shirt speaking, and a Windows taskbar at the bottom with various application icons.

Now for so that table, if we can say if we try to understand, so that table will have entries like some say some hole ids, so for a particular hole. So, it should have the information about the start address and size. So, for hole 1, the start address may be say 1000 and the size is say 15k. So, hole 2, start address is say 2000 and it is size is say 5k. So, that way it has to maintain a table likes this.

Now, you see for maintaining this table, I have to keep information about the start address and size, so both of them. And this requirement may be quite large and it may it may so happen that keeping information about 100 bytes or keeping information about 100 bytes. So, we are also wasting quite a few bytes for doing that and in many cases it may so happen, that this table entry it requires more space, than the actual free space that is available in the hole. So, that over reduce much much higher than the hole, the benefit, that we can get by keeping the hole.

So, it is very much unlikely, that this there will be a process coming in future, whose memory requirement will be less or equal 100 bytes. So, it is not worth keeping the information of this 100 bytes and so instead of allocating 14900 bytes. So, we give it 15000 bytes, the entire hole. So, this 100 bytes of space, that is there, so this process is not going to utilize, other process is also cannot make use of that. So, this is internal to a partition and it is not being used. So, this is the known as the problem of internal fragmentation.

So, external fragmentation is more serious definitely, but external fragmentation we can resolve by doing the compaction of the memory. But internal fragmentation we cannot resolve by compaction because internal for this second case this entire space is assumed to be allocated to the process.

So, we cannot just take back those 100 bytes from the process at that point of time, during compaction, so that is not possible. So, that way internal fragmentation will stay, external fragmentation can go by this compaction operation. So, this is the fragmentation problem.

(Refer Slide Time: 06:19)



Fragmentation (Cont.)

Reduce external fragmentation by **compaction**

- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time
- I/O problem -- cannot perform compaction while I/O is in progress involving memory that is being compacted.
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers

So, how can we reduce this external fragmentation? So, we can reduce by doing compaction. Shuffle memory contents to place all free memory location memory together in one large block. And compaction is possible only if relocation is dynamic and is done at execution time. So, relocation dynamic at the time of execution, the operating system finds that ok, there are many holes created in the memory.

So, it has to do a compaction, that way the process code will move from one region of the memory to another region, so relocation has to be performed. So, if the relocat[ion]-if dynamic relocation is possible, then only compaction can be utilized. Otherwise, if it is only load time relocation, so once the program has been loaded from a particular memory space, so memory location.

So, if we cannot change that start address by loading this relocation register separately. So, we have we cannot make use of this compaction. Then there is Input Output problem, IO problem. So, it cannot come perform compaction, while IO is in progress involving memory that is being compacted. So, as you know, that whenever a process write something onto some output device, so this the data does not go to the output device directly. So, it is kept in some memory buffer and from that memory buffer it is sent to the IO device later.

Now, when we are doing the compaction if that IO de[vice], this memory buffer address changes. Then naturally the content will not be accessible by the IO device. So, that way, if some IO is going on with some from some memory locations, so we cannot do this compaction process.

So, Latch job in memory, while it is involved in I O. So, we cannot move that job from the when it is doing some IO operation, so we cannot do that; so this is one solution. Other solution is, so we do IO only into OS buffers. So, we, so because operating system decides that the beginning of memory, that is not going to be compacted at all. So, OS in the OS we keep some buffers reserve for some IO devices and this IO is done with respect to those device; those buffers only.

So, in that case of course, this problem can be solved, because this compaction will not involve those buffers. But the difficulty is that the OS has to keep those buffers ready always, so even if the IO device is not going to be used. So, that buffer will be remaining in the OS space. So, that way we have got this fragmentation problem and by compaction it can be solved with some caution ok, regarding this input output operation and with the requirement, that the program so the process should be dynamically re locatable, otherwise it will not be possible.

(Refer Slide Time: 09:09)

**Non-contiguous Allocation**

- Partition the a program into a number of small units, each of which can reside in a different part of the memory.
- Need hardware support.
- Various methods to do the partitions:
  - Segmentation.
  - Paging
  - paged segmentation.

So, next we will be looking in to the other possible solutions. So, other possible solutions, that is you do not keep the process contiguous ok. So, far whatever we have discussed, whatever scheme we have discussed whether it is single contiguous allocation or this partitioning and all. So, ultimately the process is remaining in a single contiguous chunk of memory.

So, this entire chunk is given to process p1. So, the entire address phase of process p1 belongs to this particular chunk. So, the other possibility is that suppose, I have got this memory and when I take a snap shot of this memory, I find that there are some holes available. So, this is hole 1 and then this is hole 2 ok, some holes are available.

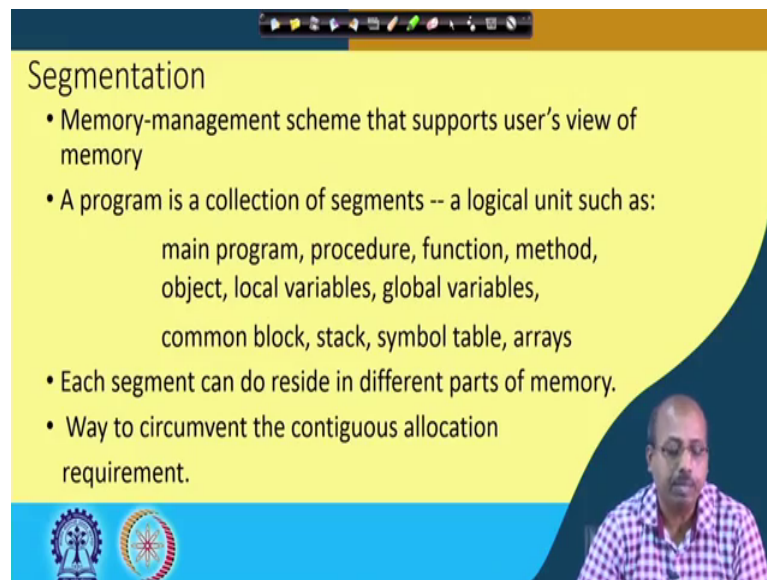
So, if it is possible, that I have got different parts of p1, loaded into different holes. Some part of p1 goes here, some parts of p1 goes into hole 2, some part of p1 goes to hole 3. So, if that is possible; so if my system can handle this type of distributed memory locations allocated to process. So, if that is possible then we can take help of this memory management 1.

And possibly utilize all these that are there in the system. So, even if my size of p1 is not more than not less than this individual holes, so if that total size of the process is less than the total size of this holes available. So, some of this holes, so if you if that the total the process size is less than that total in that case it is possible to allocate that.

So, this noncontiguous allocation is actually talking about these strategies. And there are many noncontiguous allocation policies that will see. So, partition the program into a number of small units and each of which can reside in a different part of the memory. So, as far as the user is concerned, so user can understand locate this is my procedure 1, this is procedure 2, this is procedure 3 etcetera. Now, when I am referring to procedure 1, so may be if the procedure 1 is located in a small portion, contiguous portion that is sufficient. So, it may not be necessary that both the procedures are in the contiguous locations.

So, that way I can have some idea about partitioning a program into number of small units. Definitely, it will require hardware support and there are various methods to do partitions. One is called segmentation, then there is paging and there is paged segmentation. So, will see this strategies one by one. So, each of them it will what it will essentially do is that it will partition the source program into regions. And each region will be loaded into memory independently of the other regions ok. And this as the memory management unit it will provide us the facilities by which those distributed units can be accessed.

(Refer Slide Time: 12:25)



**Segmentation**

- Memory-management scheme that supports user's view of memory
- A program is a collection of segments -- a logical unit such as:
  - main program, procedure, function, method,
  - object, local variables, global variables,
  - common block, stack, symbol table, arrays
- Each segment can do reside in different parts of memory.
- Way to circumvent the contiguous allocation requirement.

The slide features a yellow background with a dark blue curved shape on the right side. At the bottom left, there are two circular logos. At the bottom right, there is a small video inset showing a man with glasses and a checkered shirt speaking.

The first policy is known as the segmentation policy. So, it supports, so this is actually going by the users view of the memory. So, as I was telling that a program may be a collection of segments logical units such as main program, procedure, function, method,



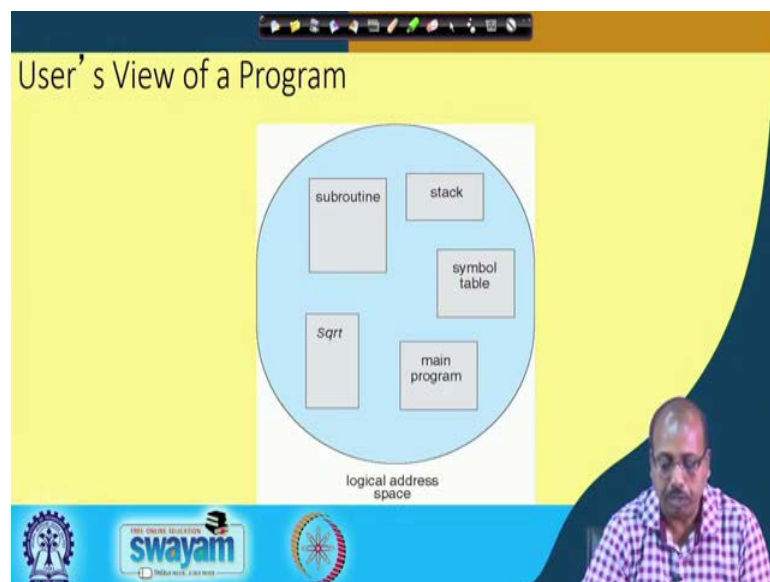
object, local variables, global variables, common block, stack symbol table, arrays etcetera. So, these are the components of a program or process.

So, as a user of the system, user of the or the developer of the program, we can visualize the program to be consisting of these components. There is a main program there is there are set, there is a set of procedures and there are set of functions, some of or object oriented paradigms, so we have got some methods objects, then some variables etcetera.

So, this way and also we understand that there will be a stack, because the process will while executing, so it will be making some procedure call and all. So we need to remember the stack, so that way these are the components of a program or process. Now, each segment can do reside in different parts of memory. So, this segmentation policy what it what it tries to advocate is that each part or each segment can be located differently in the different parts of memory.

So, this is one way to circumvent the contiguous allocation requirement, because each segment is loaded in different part. So, that way we can be able to allocate the program enough space, though not contiguous in a noncontiguous fashion will be able to allocate it space.

(Refer Slide Time: 14:09)

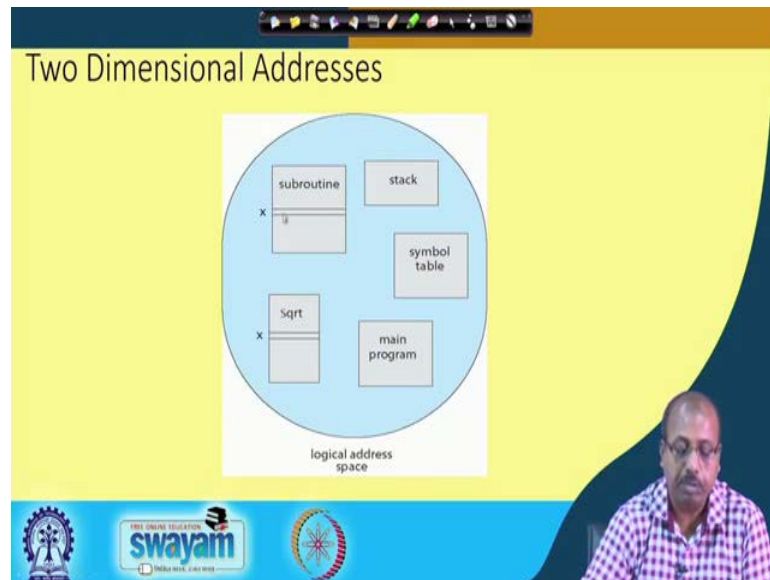


So, this is the user's view of a program. So, total logical address space that we have, so it may be it can be thought about consisting of a set of address set of components, like we



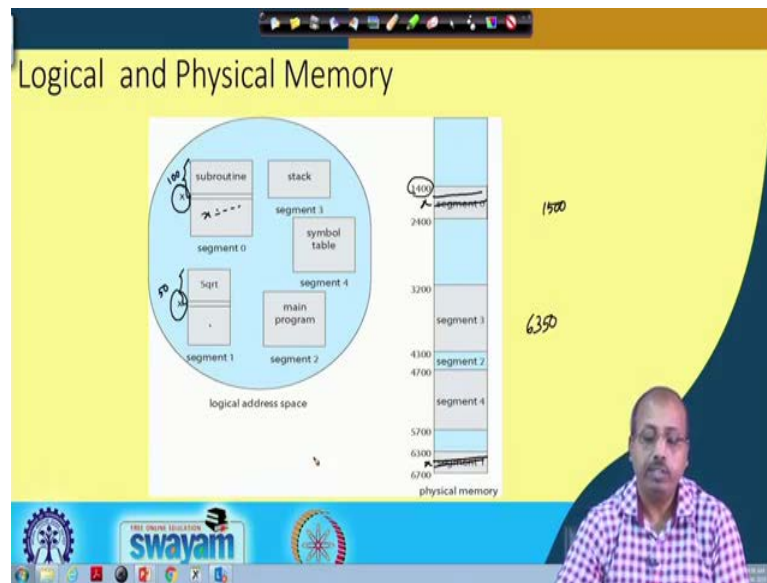
have got the main program we have got a function sqrt, then we may have some subroutines, then the stack is there, symbol table is there. So, symbol table will have all the global and local variable entries. So, that is the end from the users view point a program looks like this ok. So, it has got several segments.

(Refer Slide Time: 14:45)



Now once we have these segments, so we can we can have this another possibilities we have got 2 dimensional addresses. So, that is some part of it may be distributed like the subroutine. So, there may be 2 subroutines, similarly this sqrt, it may have a function it may have some variable x here. So, that way we can have 2 dimensional addresses.

(Refer Slide Time: 15:07)



So, basically whenever we are talking about this address space, so this whenever we are talking about a variable  $x$ , so that is there in the subroutine and there is there is another there is also another variable  $x$  in the sqrt. So, this  $x$  is different from this  $x$  because this  $x$  is local to this subroutine and this  $x$  is local to the sqrt function.

Now, if we say that these are my segments. So, this a subroutine is segment 0, sqrt is segment 1, then this is segment 2. So, that way from this user's view point, the structure of the program, so that is divided into number of segments. Now, when the program is loaded into the memory, so they are loaded following this concept of segments.

So, these each segment is loaded separately. So, may be this segment 0 that is this subroutine part, so it is loaded in this address region 14100 to 24100. Then this part is free then the again this was this space was available and this segment 3 was loaded there. Then segment 2 is loaded in this part 4300 to 4700. Then this 4700 onwards, so we have loaded segment 4.

Again there is some free space and then this segment 1 is loaded at this space. So, wherever we find enough space available. So, it may be a first fit, best fit, worst fit whatever policy that we had previously for partition memory management. So, that those policies can be added to and but of course, with the thing that now we are talking in terms of size of the segments, not the size of the entire program. So, that way we can find out enough space for individual segments.

So, wherever you find a contiguous space sufficient for a segment the segment can be loaded into that space. So, it gives us flexibility in the memory management. So, we do not have to; we do not have to give the all the segments together in the memory. Now, you see that the logical address that we have, so for example, this x and this x, so they are different. So, this x, when I am talking about so this is the x of segment 0 and this x is the a variable of segment 1. So, when I am referring to say this x in my code somewhere here so x equals to something. So, I have to access the x somewhere here. So, this may be the location corresponding to x.

Similarly, when I am talking about this sqrt function which is loaded here, when talking about the x here may be I will be talking about this x. So, while this x is being accessed, so it says from the beginning of the subroutine. So, it is at an offset of say a 100 on the beginning of the subroutine and this x may be from the beginning of this sqrt function, this is at offset x, offset 50.

Now, if I know that this subroutine starts at address 14100. So, with that we can add this logical address 100 and come to the physical address 1500. Similarly, as we know the start address of this sqrt is 6300, so with that we can add this logical address 50 and then I can get the value 6350 as the physical address of the variable x. So, that way with the logical address to physical address translation, so they will be done by adding the start address of the individual segment. So, that will see how this translation is done and the hardware support that will be needed for this.

(Refer Slide Time: 18:47)

### Segmentation Architecture

- Logical address consists of a two tuple:  
 $\langle \text{segment-number}, \text{offset} \rangle$
- Need to map a two-dimensional logical addresses to a one-dimensional physical address. Done via **Segment table**:
  - base** – contains the starting physical address where a segments reside in memory
  - limit** – specifies the length of the segment
- Segment table is kept in memory
  - Segment-table base register (STBR)** points to the segment table's location in memory
  - Segment-table length register (STLR)** indicates number of segments used by a program;  
segment number  $s$  is legal if  $s < \text{STLR}$

Segment Number	Base	Limit
0	10K	15K
1		
2		

So, the segmentation architecture that will follow, the logical address now consists of a two tuple; one is the segment number and other is the offset.

(Refer Slide Time: 19:03)

### Logical and Physical Memory

Logical address space:

- segment 0: subroutine, stack
- segment 1: Sqrt
- segment 2: main program
- segment 3: symbol table
- segment 4: main program

Physical memory:

- segment 0: 1400 - 2100
- segment 1: 2100 - 3200
- segment 2: 3200 - 4300
- segment 3: 4300 - 4700
- segment 4: 4700 - 5700
- segment 1: 5700 - 6300
- segment 1: 6300 - 6700

Handwritten notes:

- $x-3 (0,100)$  points to segment 0, offset 100.
- $x-3 (1,50)$  points to segment 1, offset 50.

Like as I was telling that this x is of this is of this is at offset 100. So, it will, so this x will be told in terms of a tuple which is segment number 0 and offset 100. Similarly, this x, so this x will be told to be x this x is the segment number is 1. And the offset is say this is say 50 as I was telling previously, so 150.

So, any logical address generated by the CPU. So, this will be having 2 parts; one is the segment number, other is the logical the offset from the beginning of that segment. So, this way we will have this logical address it will consider two tuple, segment number and offset. And need to map, a 2 dimensional logical address to a 1 dimensional physical address and this is this ultimately, when it comes to memory. So, memory does not understand this segment number and offset. So, memory chip will need the full address bits coming.

So, there has to be a transition. So, this value so we can understand that so this say this is say 2 dimensional quantity, because we have got this segment number and offset. So, there are 2 quantities, so that way it is 2 dimensional quantity. So, it has to be converted into a 1 dimensional physical address and this is done via segment table. So, segment table it has got two entries, one is called base and other is called limit.

So, this is basically we have got this is the segment table. Now, for the first segment or segment 0, I will have two entries the base the start address and the limit. So, it says that this base is say 10k, that is a segment starts at 10k and a size of a segment may be say 15k. So, when this when this logical address is given, so the we have got all these entries of segment 0 segment 1 segment 2, so we can have all these entries. Now, whichever segment is talked about, so that segment number can be used to index into this table and then from there we can find out the base address we can find the limit. Now, this limit value will be compared with this offset to see whether the address generated the offset generated is within the limit or not.

So, we have got at every entry of this segment table, one base value and a limit value. So, that specifies the length of the segment. Now, the segment table, so this particular table that we are talking about so that is called the segment table and this segment table should be kept in the memory. And then there is a segment table base register that points the segment tables location in memory.

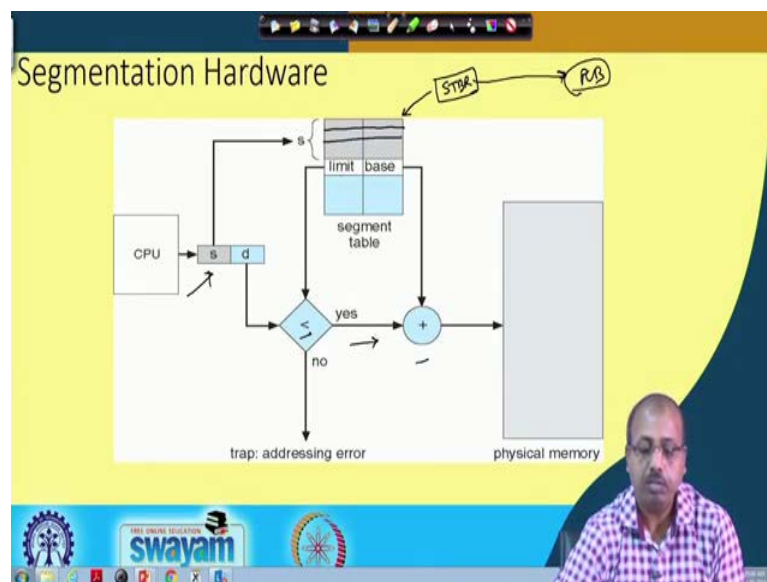
So, what I am what we will telling is that suppose this is the full memory, now some part of the memory has to be dedicated to keep the segment table for the process. So, if this is the in this part suppose I have kept the segment table. So, this is the segment table and in this segment table we have got the entries like this say this base and limit.

So, this is the segment 0, base and limit then the next 2 entries are the for the segment 1 base and limit. So, like that we have got the entries for the segment table. And this segment table start address is say 5000 fine. So, that is the segment table base register, so it will contain this segment table start address 5000 and there is a segment table length register that will indicate the number of segments used by a program.

So, for this particular example suppose we have got say 3 segment. So, this STLR will be equal to 3 and STBR will be equal to 5000 and then this segment number  $s$  is legal if  $s$  less than STLR. Because any segment addresses that is generated segment number. So, if it is less than STLR, then only the segment number is valid. Then we can we know that this starts at STBR.

So, we can go to the corresponding entry the first 2 entries are for segment 0, next 2 entries are for segment 1, next 2 entries for segment 2, so like that. So, we can come to the appropriate entry in this table in this segment table and then from this table we can extract the base and limit and check with the corresponding offset and we can add that base address to the offset to come to the actual physical entries.

(Refer Slide Time: 23:45)



So, we will see that how this translation is done, so this is the thing. So, we have got; we have got this situation that this CPU it has generated a logical address in terms of it has generated a logical address, it has generated a logical address in terms of this segment number and this offset  $s$  and  $d$ . Now, this segment table is accessed and the segment

table this  $s$  is used to index into the segment table and as I said, so it has got in it is like this limit and base like that. And this start address of this segment table is available, in the STBR register, which is not shown in the diagram.

So, this STBR register in turn, so you can understand this STBR will be contained in the process control block of the process. So, this process control block if you remember, so there was an entry called memory limits. So, that memory limits information, so that is in this particular case. So, this PCB will contain that STBR address for the particular process, in that memory limit entry.

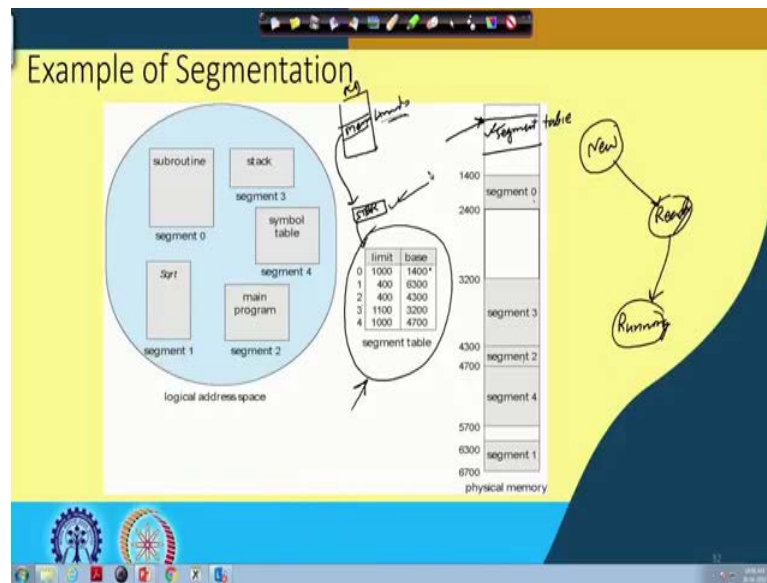
Now, this STBR gives me the start address of this of this segment table and then with that start address this  $s$  will be added. So, to come to the appropriate entry in the segment into the segment table, suppose it comes to say this particular entry containing the limit value and the base value. Now, this limit will be compared with the offset  $d$ .

Now, if this say  $d$  is less than the limit then only if  $d$  is less than the limit then only it is a valid logical address. Otherwise, it is trying to access some other location which is not within the current segment. So, that should be stopped because that is some sort of segmentation, violation problem you might have come across this particular thing in some programs while running on some operating systems. Particularly Unix and Linux operating systems. So, this segmentation violation, so it is actually checked from here, so this  $d$  is less than limit or not. If  $d$  is less than limit then it is fine the address is ok.

And now this offset  $d$  will be added with the base and then it will be access in the corresponding physical memory. So, this is the overall address translation scheme, now the this memory management unit that we have, so it must satisfy must support this particular operation. So, it should support this check it should be able to divide this address into  $s$  and  $d$  part. Then this has to generate it has to have this check and addition part, so all this had to be done. So, will see how this can be carried out in the context of previous example.



(Refer Slide Time: 26:45)



Suppose, a previously we had got so many segments, segment 0, 1, 2, 3, 4. So, segment, so this is the segment table register. So, segment table register, so this start address of this segment table registers. So, this is stored in the STBR, so again it is not shown here explicitly and this STBR will have this thing and this entire segment table is stored somewhere herein, so this is the space for storing the segment table. And this STBR register is actually pointing to this ok.

So, memory management unit it will have the STBR register. So, when the process is made to run. So, it is making a transition from ready to run at that time when the context switching occurs then from the PCB of the process. So, this is the process control block of the process and there are there is an entry called memory limits.

So, from this STBR value will be loaded. So, this is a register within the memory management unit of the processor. Now, this STBR, so this has this segment table has got these entries. Now, as I was telling that the segment 0 is loaded from location 14100 ok. So, you see the corresponding base value has been noted and the size of the segment is 1000 bytes. Similarly process sorry segment 1, it is loaded from location 6300, so 6,300 plus size limit is 400. Similarly segment 2, it is starting at location 4300 and it is a size 400.

So, that way this particular table has been formulated once the program has been loaded. Now, basically when this when the process was making a transition from new to ready,

when we when it was making a transition like this at that time the loader actually decided found out what are the available memory for free spaces and it had found that these are the free spaces where the segments may fit ok. Accordingly it loads that corresponding segments into this free holes and fills up this segment table here ok; fills up the segment table here and in the PCB of this process its notes down the STB, it notes down the start address of this segment table.

Later on, when the process makes a transition from ready to running so at that time, what the system does is that it just copies from the PCB that STBR the segment table start address value kept in the memory limits register memory limits entry into the STBR register of the process. So, that way it operates and we this entire address translation is done accordingly.

So, this is a quite complex process as you can understand and this process switching then this process loading, process switching, new to ready, ready to run so all this transition that are occurring. So, that is definitely some role to be played by the memory management unit. So, you have to think a whole thing in a total fashion ok. Look into the total situation together to understand like how this whole thing is going to operate. So, we will continue with this in the next class.