

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

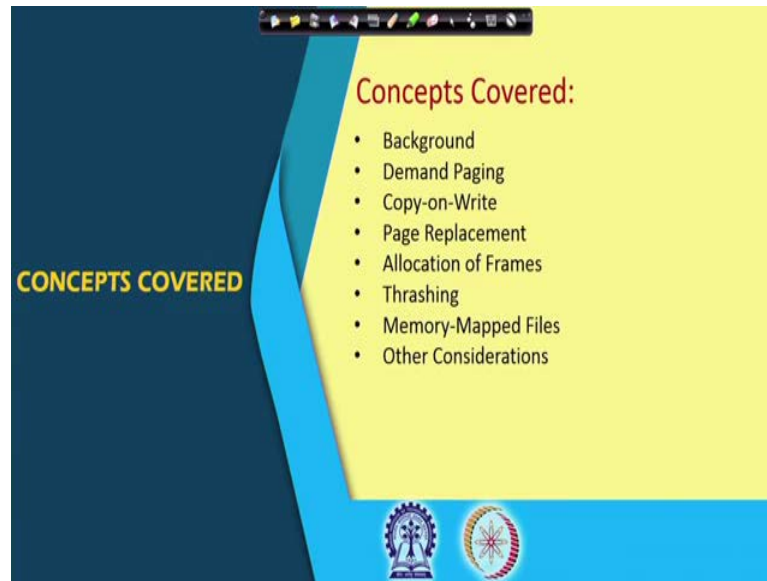
**Lecture - 49**  
**Virtual Memory**

Next, we look into the idea known as Virtual Memory. So, this has become one of the very important concepts in the memory management policies, that are followed in these advanced operating systems. And today almost all the operating systems that we have on general computers, they follow this virtual memory policy, because this literally relieves the user from this limitation of the main memory. So, even if my main memory size is less, so I can think about writing programs which are pretty large in size much larger than the actual physical memory that is available.

So, the concepts that, we have discussed, so far, till the last class, so there we have seen that we can have this programs written. But when a program or process starts executing, the entire process has to be loaded into memory, before it executes. So, the entire process should be loaded into memory. So, if my memory size is say 64 GB, then I cannot have a program which is larger than 64 GB, though 64 GB is a very big number, but even then.

So, normal systems if I have got a memory in the range of say megabytes then, I cannot think of programs which are of size in the gigabytes to be executed on that. So, this virtual memory concepts, so this will help us in a doing that thing. So, it will allow us to run programs, which are much larger than the actual physical memory size.

(Refer Slide Time: 01:54)



So, the concepts that we are going to see are, the Background, first of all the background material than the demand paging. So, demand paging as the terms suggests, so it means that on demand the pages will be loaded. So, if there is requirement that a page has to be accessed by the CPU, then that page will be loaded otherwise it is not. So, that is the beauty. So, this actually helps us, because many times what happens is that a program may have large number of pages, but many of those pages are not required, particularly the error handling routines and all.

So, in normal execution of a program, so it is very very much possible that, this error condition does not occur. So, having this error control module also loaded into memory. So, this is nothing, but wastage of memory. So, if we can do something on that line, then the copy on write. So, whenever we are modifying something, then we have to write it. So, let us we look into this concept, then the page replacement. So, page replacement concept means that at any point of time suppose, I have loaded four pages of a process and now while executing the process makes a reference to page number five. So, which is not there, in the memory?

Now, it may, so happen that, all the memory frames are occupied. So, none of them are free, then we have to find out some frame such that the corresponding page is not being refer to at present. So, that page, we can copy back to the secondary storage and make

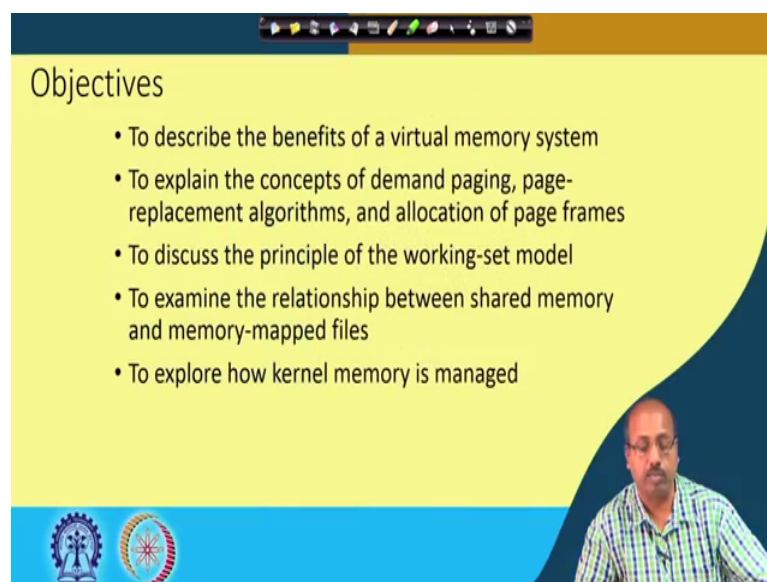
space for this new page number five for the process. Now how to find this page to be replaced? So, that comes under the broad heading of this page replacement.

Then, how many frames do you need to allocate for a process? So, ideally the answer is zero, I initially do not load any page for a process. But as and when the process starts executing, so it will be, it will generate some error, that page is not present, then it will be, the corresponding page will be loaded.

Now, this particular, the process is sometimes questionable because of this processor instruction set. We will find that we need to allocate a certain number of frames to the processes. Then there is a concept of thrashing which says that if a page is spending most of the time in a swapping in and swapping out of pages. So, that is called a thrashing situation.

Then memory mapped file, so this that that concept is there some of sometimes we want to treat this memory locations as files. So, file handling is accessing the secondary storage. So, instead of that, so if some part of the memory it self can be used as file. So, that helps in the operation of this virtual memory and other file handling applications and the other considerations that we may have in the in the context of virtual memory.

(Refer Slide Time: 04:59)



Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model
- To examine the relationship between shared memory and memory-mapped files
- To explore how kernel memory is managed

So, the objectives are like this; so we want to describe the benefits of a virtual memory systems, that the first thing that we should understand why virtual memory is needed.

Second thing is that, we will be looking into the concepts or demand paging page replacement algorithm and allocation of page frames, and then we will be looking into another very interesting concept known as working set. So, working set means that the minimum set of pages that a process will need at some point of time for it is smooth execution. So, may be if a program, if we considered the program to be a very almost sequential program, then at the initial part of it is execution. So, it will be referring to the statements or instructions that are there in the initial part of the process. So, that is, that I can say that is the page 1.

Now, after sometime it will be going to refer to other pages. So, in that way if you look into this process at a particular time, we will find that only some of the pages of it are being refer to at this point. And if you take a snapshot of the system again after sometime, we will find possibly a different set of pages are being refer to at that time.

So, this set of pages that are being referred to at a particular time. So, this will form the working set for the process. So, this working set is a very important concept and that on that we have got this virtual memory concept being successful. Now, to examine the relationship between shared memory and memory mapped file. So, we will be looking into that and then we will see how the kernel memory is managed in a virtual memory environment.

(Refer Slide Time: 06:38)

**Background**

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running; hence more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory; hence, each user program runs faster

for  $i=1$  to 1000  
for  $j=1$  to 1000  
 $a(i) = b(i) + c(j)$

52

The slide features a yellow background with a blue footer. In the bottom left corner, there are two circular logos. In the bottom right corner, there is a video inset showing a man with glasses and a mustache, wearing a green and white checkered shirt, speaking. The slide contains a list of bullet points and handwritten notes in black ink.

Now, to start with as a the up to the memory management policies that we have seen up to last class, where we have looked into this different memory management schemes like contiguous to partitioned to segmented to paged. So, the code needs to be in memory to execute, but entire program is rarely used. So, as I was telling that in the entire program. So, there may be some procedures which are not being used at they are not being used regularly.

For example, some error code ok, some error condition occurring and then a corresponding error handlers that we have, then unusual routines sometimes maybe, you have got some condition and if some exceptional condition occurs, then only some routine is called for. So, maybe in some calculation in some computation, there are some special cases of computation and only when the special cases occur the corresponding routines need to be invoked. So, that is the problem

So, then if we are going to large data structure, like you say, you see that, if you have got an array of all the employees data of an organization. And now if you are trying to modify that data set, then what is happening is the at one point of time you are just referring to some of the employee records not all the employees together.

So, that way we have got, if the data structure is very large, then we can they think about; then we can think that at some point of time only a part of the array is being manipulated. For example, if I have got a piece of code say for,  $i$  equal to 1 to 100, for  $j$  equal to say, 1 to 1000. So, if we say  $a[i] = b[i] + c[j]$ ; so like this. So, you see initially we are the just looking into first thousand entries in a array, first thousand entries in b array, first thousand entries in c array after sometime, we are looking into the second thousand bits of a b array, so like that.

So, when the when this a first part of execution is over, first thousand elements have been processed, then possibly I am not going to use it in near future. So, this if, the data structure is pretty large then it may, so, happened that at any point of time computation is restricted only in some part of the data structure it does not affect the entire data structure. So, we do not need to have the entire data structure together in the memory.

So, entire program code is there not needed at a same time. So, this is the first observation as I was telling that entire program code. So, once we start maybe, we can start only with the main routine. And from the main routine, it is giving called to some

procedures then that procedure code is needed and once we return from that procedure that code is no more needed. Maybe next time the main routine will be calling some other procedure in sequence.

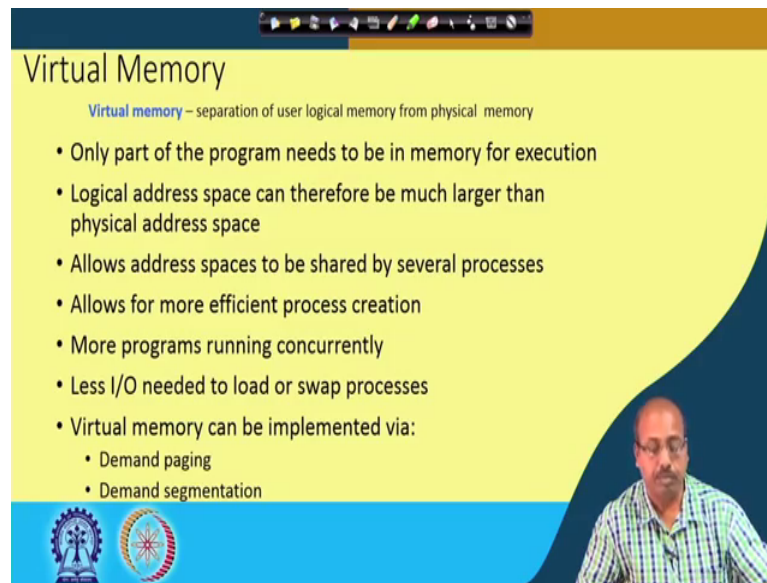
So, the entire program code not needed at same time, consider the ability to execute partially loaded program. So, this is the idea that the virtual memory will hinge upon. So, we will be loading a program partially and a based on that will be trying to execute that program.

So, program no longer constrained by limits of physical memory. Now, since I have got partially loaded program, so even if, I get only a few frames free in the memory, then I can load few pages of the program and start executing it. So, in the boundary case, so I need only a single frame to load the first page of the process and that way it can start. So, each program takes less memory while running, hence more programs can run at the same time. So, this is the good thing because I am not loading the entire program. So, a program takes fewer amounts of space and so running the program. So, that can help us to run the program more number of programs.

So, CPU utilization will increase and a throughput will that will also increase with no increase in the response time or turnaround time. So, response time, turnaround time they will not increase, but this CPU utilization and throughput will increase less I O will be needed to load or swap programs into memory. Hence each user program runs faster, since I am not trying to load the entire program or entire process at the beginning.

So, my the transition from this new to ready, so that can be pretty fast, I do not load the entire program into the memory to make it to a ready state. I just load first few pages and make the program make the process ready for execution. So, that way less I O will be needed to load or swap programs into memory. So, each user program will definitely run faster because the response will be coming faster.

(Refer Slide Time: 11:52)



The slide is titled "Virtual Memory" and features a yellow background with a blue wave-like shape on the right side. At the top, there is a navigation bar with various icons. Below the title, a subtitle reads "Virtual memory – separation of user logical memory from physical memory". A list of bullet points is centered on the slide, and a video inset of a man in a checkered shirt is positioned in the bottom right corner. The bottom of the slide contains two logos: one of a gear and a person, and another of a gear and a sun.

## Virtual Memory

Virtual memory – separation of user logical memory from physical memory

- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical address space
- Allows address spaces to be shared by several processes
- Allows for more efficient process creation
- More programs running concurrently
- Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

So, this with this background, so we come to the concept of virtual memory. So, it separates the user logical memory from physical memory. So, only part of the program needs to be in memory for execution. So, as I was telling only first few pages are loaded and logical address space can therefore, be much larger than the physical address space.

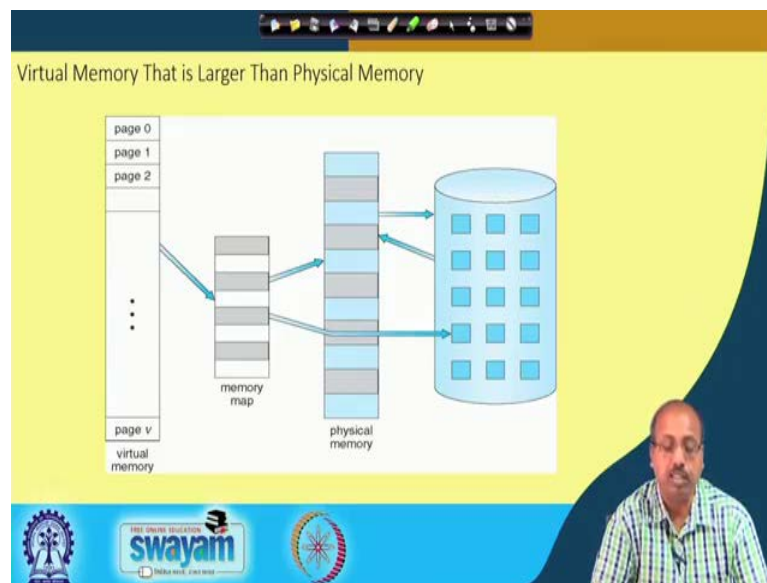
So, now logical address space can be very large. So, it is not restricted by the size of the physical memory and it allows address spaces to be shared by several processes. So, this address space, the same address space can be shared by number of processes allows more for more efficient process creation. More programs can run concurrently and less I O needed to load or swap processes; So, this is the idea.

So, virtual memory can be implemented by two mechanism, one is called demand paging another is called demand segmentation. So, both the cases the idea is very simple. So, if a particular portion of memory is required to be accessed by a process, then only that portion will be loaded into main memory. So, for paged memory management, so if a particular page is asked for, they by the processor then only that page will be loaded into memory.

Similarly, demand segmentation, there if a particular segment is being referred to by the program then that particular segment will be loaded into memory. Otherwise they will reside in the secondary storage, that the disk only and assuming that the disk size is

pretty large and it can be consider to be infinite for the processes. So, this the process size is not restricted by that also because disk size is can be very large compare to the physical memory size. So, my process can see as if, I have got almost infinite memory for execution

(Refer Slide Time: 13:53)



So, this is the basic idea. So, virtual memory that is larger than physical memory. So, a, so, these are the virtual pages of a process. So, a process has got a  $v$  number of pages ok. So, that is; so that is a total size of the process and that is divided into  $v + 1$  number of pages.

Now, this memory map, so this will be putting some of this pages onto here, you can considered as if this pages that are, they are in the memory map. So, for some of the pages it is there in the memory and others are not. Now, when a particular page is being refer to, so it will be looking for that in the physical memory and if the page is there in the physical memory then it is sufficient.

So, it will be accessing that particular page and if the page is not there in that case, the OS we look into the secondary storage for that particular page, it will determine the corresponding block address. And from that block address that particular page will be copied into a free memory frame and then the process will assume.



So, the difficulty that we have is that for example, say this particular access. So, when the program was accessing say these particular logical page, it could find the corresponding physical page here. But it may, so, happen that at some other point when it is a trying to access this particular page, so, this page was not present in the memory. So, what it; so, it could not find it in the physical memory. So, it goes to the disk to locate the corresponding page in the disk block. And once it has located it, so, this page will be loaded into the main memory and after that only the execution will assume and it will continue.

So, you see the advantage, that we are getting is that we are not restricted by the size of the physical memory. But the price that we are paying is, that every time I have to see like what is the this extra additional time, that is coming if the page is not there in the main memory. So, I have to pay for this disk access and as you know that a disk access time is a pretty slow, compare to this memory or this CPU operation. So, this is going to take good amount of time ok. So, that has to be factored in to for program execution.

(Refer Slide Time: 16:20)

The slide is titled "Virtual-address Space" and features a diagram on the right side showing a vertical stack of memory regions. From top to bottom, the regions are: "stack" (growing down from "Max"), a large blue "hole" (unused space), "heap" (growing up), "data", and "code" (at the bottom, labeled "0"). A hand-drawn diagram on the right shows a stack of pages with arrows indicating the stack growing down and the heap growing up. The slide contains the following bullet points:

- Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
  - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation

The slide also includes logos for "swayam" and other educational institutions at the bottom.

So, if you consider the virtual address space of a process, then the usually we design logical address space for stack to start at maximum logical address and grow down while the heap grows up. So, this is; so the stack start from the maximum address and it grows down and the heaps starts at the some address and it starts growing up. So, heap grows up, stack grows down. So, that actually maximizes the address space use because some

program it may be using more of procedure calls. So, it will be using more of stack and some procedure some program, so they may be doing lot of dynamic memory allocation dynamic memory usage.

So, it may use some dynamically it make a make calls like malloc for memory allocation and then that way you makes the use of the heap storage. So,, they are given two different ends of address space. So, it maximizes address space use and unused address space between the two is the hole. So, there is for example, this blue region that we have here. So, that is a hole because this part is not has not been utilized, so far.

No physical memory needed until heap or stack grows to a given new page. So, this is, a when this thing happens, so, this is the virtual address space this is happening. So, physically may be I have allocated one page, I had initially allocated one page for the stack from the top side.

So, it was growing like this, after it has come up to this, I need to give a new physical page actually, what I am telling is that; so this entire thing is logical only, so physically this address is not given. So, initially, what we do is that the stack starts growing and or to make the stack starts growing. So, I have to allocate one memory frame for the stack. So, if this is the physical memory. So, this is allocated one frame here. So, this initial address is mapped on to say this frame.

Now this stack, whenever writing into this virtual memory, so basically you are writing on this physical memory after sometime, this block, this particular frame will be exhausted and now, I need to give a new page for the stack. So, now, we look into this physical memory and maybe we find that this is the corresponding free frame, where I can continue the stack. So, from this point onwards the sake the next virtual stack page, that actually points to this physical frame. So, this way it can grow like this.

Similarly, for the heap also it, so initially some page is allocated in the physical memory, maybe this is the page allocated here. And after sometime, if this entire space has been consumed and it asked for more then, some other frame will be found out and then this next page will be pointing to that one. So, this can happen.

So, this way this whole thing can go, can grow virtually and whenever needed then only we allocate physical page for that. So, this is the point that it is written here, that no

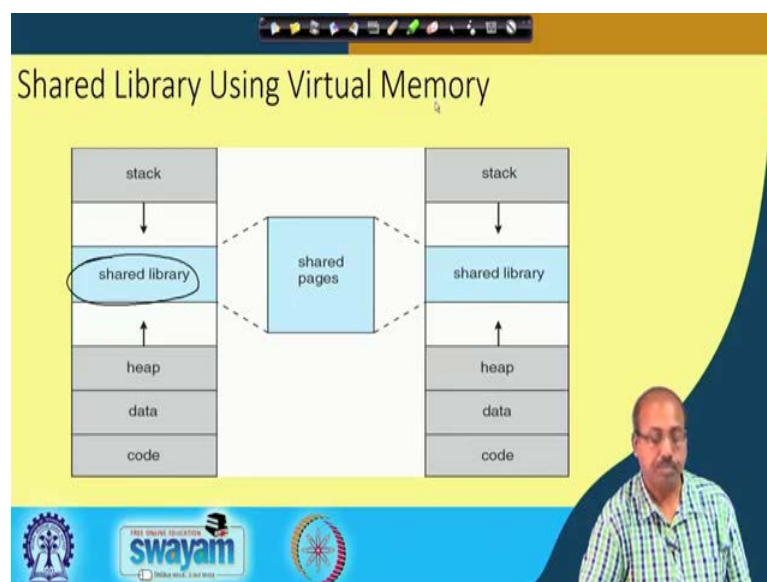
physical memory needed until heap or stack grows to a given new page. So, once the new, once the page has been exhausted, then only we need to access it new space.

So, this enables sparse addressing, a sparse address space with holes left for growth and dynamically linked libraries etcetera because of this things, so virtually the, virtually the stack is contiguous, but physically it is not. Similarly, virtually the heap is contiguous, but physically it is not, so that can happen.

So, this is, the we can has sparse address space with a holes for growth and system libraries can be shared by a mapping into virtual address space. So, several system library, so they can be mapped on to this virtual address space and there is they can be shared and shared memory by mapping pages read write into virtual address space, so that can happen.

And pages can be shared during fork, that will speed up the process creation. So, for fork I do not need to make a physically make a sharing of the pages or make a copy of the pages. So, they can be; they can be simply shared at the virtual address space level. So, physical address space level, we do not need to do anything. So, this can be done. So, that can speed up the process creation.

(Refer Slide Time: 21:04)



So, this is what we were looking into the shared library using virtual memory. So, this is the stack and heaps. So, they are going. So, out of that, so, this space is free. So, this is

virtually. So, we allocate this shared library here and that gives rise to the, so, once this shared pages are coming. So, they can be mapped onto this shared. So, shared library points to the shared pages and they can access the shared pages in this fashion.

(Refer Slide Time: 21:38)

**Demand Paging**

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed  $\Rightarrow$  reference to it
  - invalid reference  $\Rightarrow$  abort
  - not-in-memory  $\Rightarrow$  bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
- Swapper that deals with pages is a **pager**

Handwritten notes on the slide include: "m=4", "Less I/O needed, no unnecessary I/O", "Less memory needed", "Faster response", "More users", "Similar to paging system with swapping (diagram on right)", and "32 byte".

Now next comes, the concept of demand paging. So, in demand paging, so, what happens is that we can bring entire process into memory at load time. So, this is one possibility that, whenever a program is being loaded, it is making a transition from new to ready. So, this entire address space of the process, so, it is loaded into memory. So, if you have; if I have got enough number of frame available, I can copy the entire process into main memory.

Or what we can do, we can bring a page into memory only, when it is needed. So, this is demand paging is actually advocating the second policy, that is you do not load a page until and unless it is asked for, when it is asked for then only we load that page. So, to start the process, so, we can understand that only one page will be sufficient. So, it can start by loading a single page.

So, it less I O will be needed and no unnecessary I O. Unnecessary I O means that I have loaded a page and but, that is a never used. So, since demand paging it says that, only when the page will be required, it will be loaded. So, we do not have any unnecessary I O page. Memory needed, will be less, because if a process in it is lifetime, if it does not

refer to some of the pages then those pages are not loaded into the memory. So, as a result the memory requirement will be less.

Response will be faster because, because I need to load less number of pages. So, the response will be faster and definitely, we can support more number of users. Because we can, for every user, I do not have to give it, give the entire memory for that user or the all the pages of the user process to be loaded, I can only load a few pages. So, I can support more number of users.

Similar to paging system with swapping, so, this is, so, basically when a process program B is coming. So, these are the corresponding pages. So, they are swapped into it and maybe when program A, we need to claim the memory currently allocated to program A. They can be swapped out kept in some disk blocks and then eventually making that space held by program A free.

So, that is why it is called, it is similar to paging system because here also, we will be talking in terms of pages only and the swapping mechanism will be there. So, page is needed there is the reference to it and if the reference is invalid, then we abort it. And if the reference is the reference said says that the page is not in memory we bring the, bring it to the memory. So, if the page is invalid; that means, there is some problem in the reference. The page that a process is asking for is not within the address space of the process.

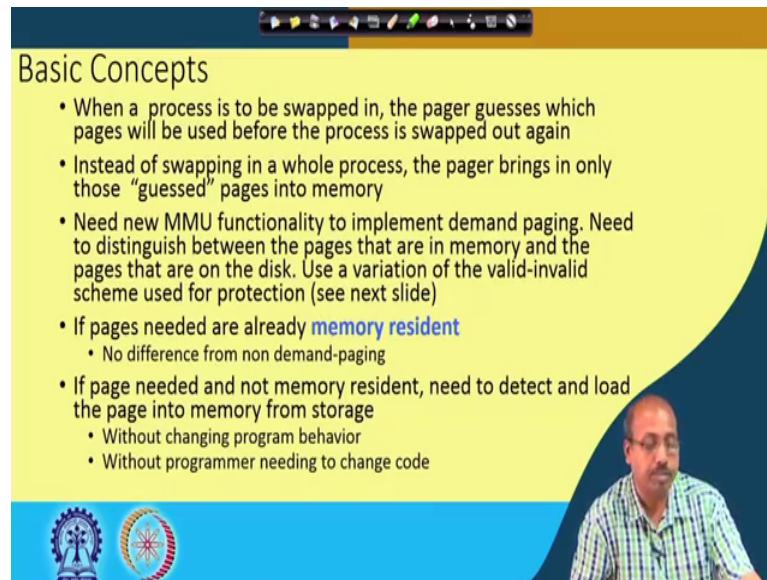
So, as a result that is an invalid reference. So, in that case the action is to abort the memory access and other possibility is that the access is valid, but the page is not present in the memory. So, in that case the system has to bring, that page from the secondary storage into memory. So, bring to memory.

And there is a concept called lazy swapper. So, it never swaps a page into memory unless the page will be needed. So, unlike the paging based policy that we have seen previously where a program or process is loaded at the beginning of execution, this lazy swapper scheme it says that only when the process will require a particular page that page will be loaded into main memory.

So, swapper that deals with a, pages is called a pager. So, this is also called a pager because it actually deals with a paging of swapping in and swapping out of pages ok. So,

this demand paging is going to be a very important criteria for or very important facility for, the supporting a higher page sizes. So, next we will be, looking into how this demand paging can work.

(Refer Slide Time: 26:00)



The slide is titled "Basic Concepts" and contains the following bullet points:

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again
- Instead of swapping in a whole process, the pager brings in only those "guessed" pages into memory
- Need new MMU functionality to implement demand paging. Need to distinguish between the pages that are in memory and the pages that are on the disk. Use a variation of the valid-invalid scheme used for protection (see next slide)
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident, need to detect and load the page into memory from storage
  - Without changing program behavior
  - Without programmer needing to change code

The slide also features a small video inset of a man in a checkered shirt speaking, and two logos at the bottom left.

So, when a process is to be swapped in, the pager guesses which pages will be used, before the process is swapped out again. When a process comes into the system, so, pager will try to see which pages will be used, before the process goes out. So, accordingly it wants to load those pages. So, instead of swapping in a whole process, the pager brings in only those guessed pages into memory. So, how this guess occurs? So, that is of course, a question.

So, maybe that it is due to history that is for a particular process, or a particular program. The system remembers that last time this program was run, which pages were they refer, were required. So, that is one possibility, other possibility is that maybe a program is running and it goes into a blocking state. When, it goes into blocking state, so, we remember, the history of the process. And from the blocking state, when it comes back to the ready state, so, at that time which pages of the process be loaded. So, that is guided by the guessed, that we have. So, instead of for loading the whole swapping in a whole process, the pager will bring out only the guessed pages into the memory.

Need new MMU functionality. So, memory management unit becomes more complex, definitely for implementing this demand paging. So, we need to distinguish between the

pages that are in memory and the pages that are on the disk. So, as I said the portion of the disk. So, that is marked as this, that is marked as the position where this the processes can be loaded. So, that makes it a difficult, because the memory management unit has to understand that some pages are in memory whereas, some pages are not there in the disk.

So, we have to use some variation of this valid invalid scheme for protection. So, previously for paging, page protection we have seen that, we have got this valid invalid bits. So, here also we can utilize this valid invalid bits to distinguish between the pages that, are there in the main memory and the pages that are to be taken from the secondary storage.

If pages are needed are already memory resident then, there is no difference from the non demand paging scheme. So, memory resident means those pages will never be removed from the memory. So, if that is the criteria then off course, demand paging are normal paging does not have any distinction they are all the same. But if page needed and a not memory resident, so, if you need a particular page and the page is not memory resident it is not permanently sitting in the memory. We need to detect and load the page into memory from storage. So, you have to figure out where exactly this page is there in the secondary storage and from there we have to take that page and load it into main memory.

Without changing program behavior, so, this is very important. So, due to this operation, so, it should not happen that the, program behavior changes. So, program operation changes and also programmer will not need to change the code programmer will it will happen a transfer and fashion. So, programmer will not be feeling that, something is going on in the system my process. So, which was running in the CPU, so, it has been taken off, from the main memory. So, it was previously loaded, but now it is taken off from the main memory all together. So, that the programmer is should not be able to understand ok. So, that it the OS design should be such that. So, these things are hidden from the programmer.

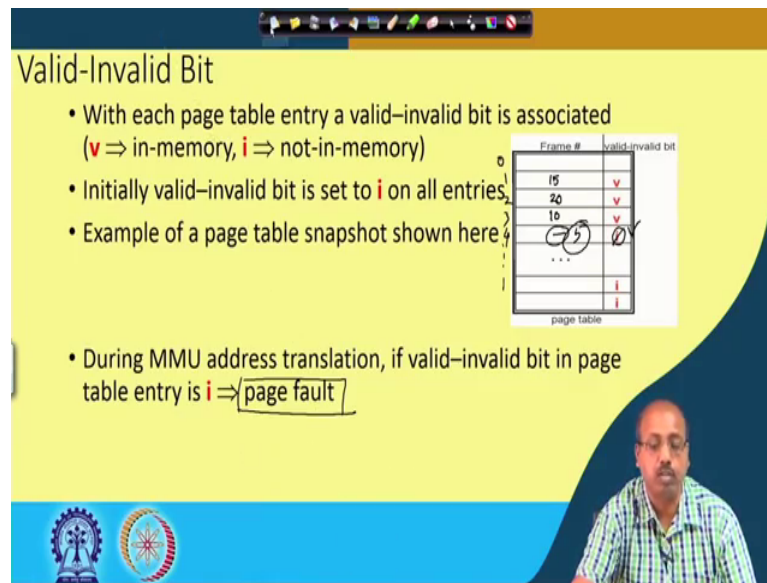
(Refer Slide Time: 29:39)

### Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v**  $\Rightarrow$  in-memory, **i**  $\Rightarrow$  not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries.
- Example of a page table snapshot shown here.

Page #	Frame #	valid-invalid bit
0		
1	15	v
2	20	v
3	10	v
4	5	v
...	...	...
...	...	i
...	...	i

- During MMU address translation, if valid–invalid bit in page table entry is **i**  $\Rightarrow$  page fault



So, you have got this valid invalid bit. So, with each table entry we have got a valid invalid bit. So, v the valid invalid bit, if it is equal to v maybe 0, so, it is or 1, so, it is in memory. And i is invalid, so, it is not in memory. So, initially all these valid invalid bits are set to I, for all the entries and example page table is here. So, for the first for some of the pages you see that the valid bit is equal to v.

So, those pages are present in the maybe corresponding memory frame. So, what happens is that. So, this is the page table and this is page table index 0 1 2 3 etcetera then, out of that we see that page number 1. So, the corresponding frame number maybe 15 and it is valid similarly this frame number maybe 20 and this is also there. So, this maybe say for 10, but this is i. So, here I do not need to mention anything. So, whatever is there, so, that is an, that is a garbage.

So, whenever a process accesses the page table ok. So, it has to first see that whether it is the valid invalid bit is set to valid or not. So, if it is not so. In that case, the page is not there in the main memory. So, it has to bring that page from the secondary storage into main memory and the after that only it can access.

So, after that this i will change to v. So, this value will have some proper value say 5, if the corresponding page is loaded at frame 5 and then it will be using this value to access the corresponding memory frame.



So, during MMU address translation if valid invalid bit in page table entry is 0. So, this is called a condition which is known as page fault. As if that is, this is a very important concept this page fault will be referring to this again and again in our discussion in this part. So when a page is referred to and the page is not present, in the main memory, so, that is a page fault. Because when this page fault occurs, the page has to be brought from the secondary storage.

And as we know that the secondary storage is very slow, so, it takes time. So, the whole issue is, how to reduce this number of page faults? And when a page fault occurs how to reduce the total amount of time needed to service this page fault? So, in our successive lectures. So, we will be looking into this concept. So, again and again and see some solution to all these issues, we will continue with this in the next class.