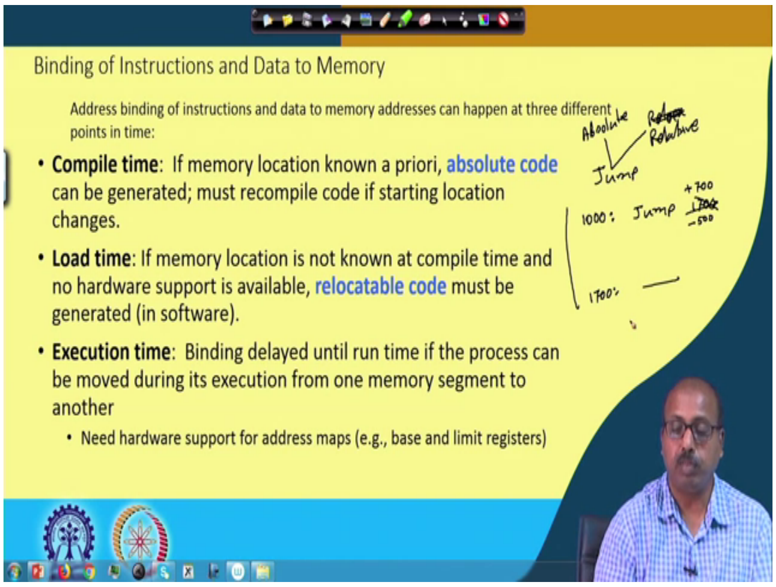
Operating System Fundamentals

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

Lecture - 42
Memory Management (Contd.)

(Refer Slide Time: 00:28)



Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different points in time:

- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
- **Load time**: If memory location is not known at compile time and no hardware support is available, relocatable code must be generated (in software).
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
  - Need hardware support for address maps (e.g., base and limit registers)

So binding of instructions and data to memory so this becomes an important issue because we need to tell like what is the physical address for each of the data variables that we have in our program and what is the actual address at which they are mapped to. So, address binding of instructions and data to memory address can happen at three different points in time.

First one is the compile time. So if this binding is done at the compile time, then it is called an absolute code. So, it may so happen that at the compile time we know that this program will be loaded from memory address 10000. So, the compiler can generate code assuming that start address of the program to be 10000. So, previously we say it that the compiler generates code assuming that the module starts at address 0. So, instead of that if the compiler assumes that the code will start at 10000 so, there is nothing harm. So, if the code that is generated so, if it is loaded from location 10000 it will work fine.

However, the difficulty is that every time you want to run the program it must be loaded from location 10000 otherwise the program will not work. Apparently it seems that why this thing be done because this is because it is making the system non-flexible. But, in many cases so, it is not necessary like a particularly for embedded applications so we know the programs that will be there in the system and as a result we can do a memory mapping of all those programs separately and before the system comes into existence so, we can determine the memory address for each of those programs.

So, every time the program is loaded so, the it may be loaded at the same space. So, we do not need this different addresses for different invocations of the program. So, we at compile time itself so we can decide that we have got an absolute address for all the variables and the absolute code is generated. So, if we are changing the starting location means you have to recompile the program. So, it is a must be recompile to if the starting location changes.

Then this address binding of this instruction and data variables may be done at the load time. So, the memory location is not known at compile time and no hardware support is available, relocatable code must be generated. So, if it is so, if we have got code in such a fashion that we do not know like from where the program will be loaded and then what is the hardware support that is there is no hardware support for doing this address corrections and all, then we have we must have the code relocatable.

So, in that case if you look into this machine instructions then you will find that there are several types of jump instructions. For example, if you are at location 1000, and you are if you are going to jump to say location 1700; so you can write jump 1700, so that the next execution starts at 1700. Now, this particular code is not relocatable code because this if the program is loaded from a different address, then all these addresses will change. This jump 1700 should not be 1700.

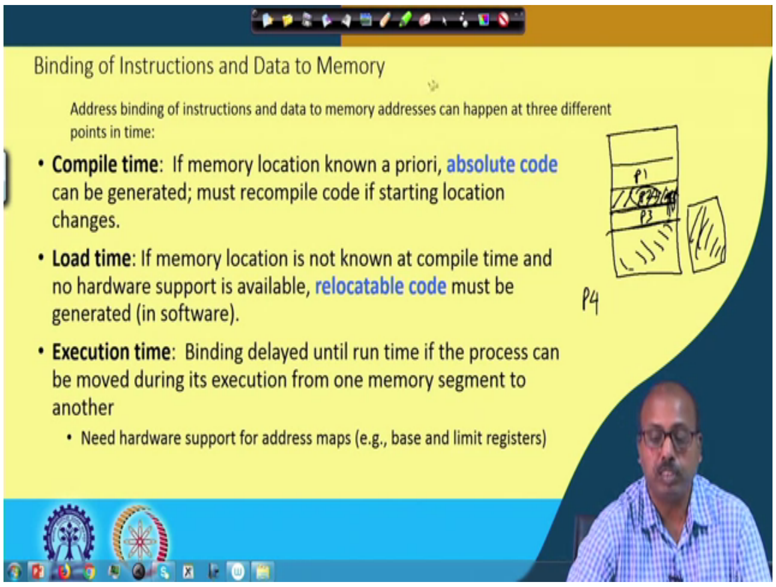
But, if I say that this jump is not a not 1700, but plus 700 that is whatever be the current location with that 700 from that it has to go forward by 700 locations. Or, if I say it is minus 500 that is whatever is the current location from there it has to go back by 500. So, in that case we have got this program that the code that is generated it is a relocatable code because it does not matter where the program is loaded.

So, this jump and many processors you will find that there is an absolute jump, there is an absolute jump and there is a relocatable jump or sorry, it is called a relative jump. So, this relative jump it actually tells with respect to the current position where to go, but by how much offset the next jump be done. Or if you are talking about a variable access the address of data variable then either you can tell the absolute memory address where the variable is located or you can tell from the current location how far is the variable. So, that way also we can tell the offset. So, they are called relative addressing.

So, if you are following relative addressing then the code that is generated is a relocatable code. So, at load time so, if it is a relocatable codes then it does not matter. So, wherever you are loading the program it is relocatable code if we have, then it does not create any problem.

Then this binding can also be done at execution time. So, binding is delayed until run time if the process can be moved during its execution from one memory segment to another. So basically it may so happen that during execution, so the program is moved from one part to other. So why this should be this may be needed?

(Refer Slide Time: 05:44)



Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different points in time:

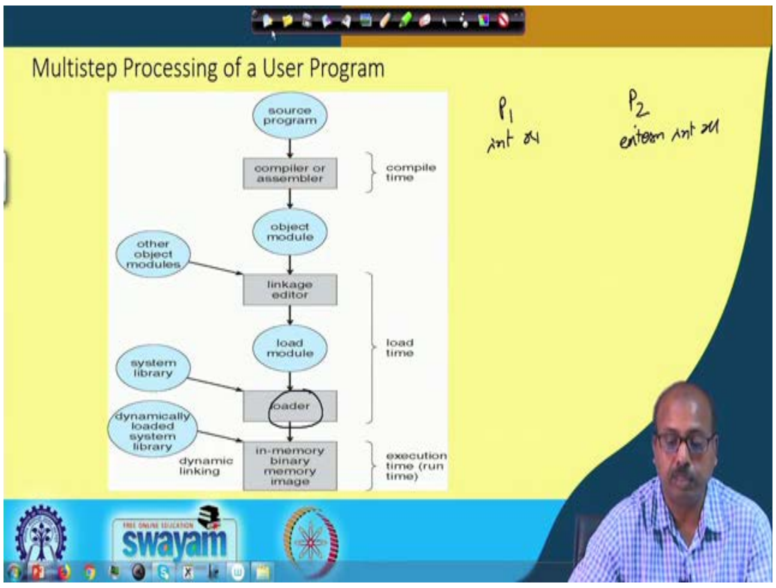
- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
- **Load time**: If memory location is not known at compile time and no hardware support is available, relocatable code must be generated (in software).
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
  - Need hardware support for address maps (e.g., base and limit registers)

So one typical situation is like this that suppose I have got this is the full memory that we have and there are multiple user programs that are running. So there are multiple users. So, this is the process P 1 is running, so after that process P 2 is there, then after that P 3 is there. Now, what happens is that after some time suppose P 2 finishes so that this block becomes free. Now a new process P 4 comes and P 4 requires a high amount of memory. So it requires say this space as well as a part from here, but this P 4 has to be given a contiguous memory, but since the none of the two region where we have got this memory available so they are not sufficient for the requirement of P 4.

So, in that case we may like to do something called a compaction. So, what we do we move this P 3 up; we move this P 3 up, so that this hole that we have here comes down. So, as a result we get a bigger chunk available in the memory in the lower side. So, then that can be given to P 4. So, what is happening is that as far as the process P 3 is concerned so it was previously in this region due to this compaction it has moved to a moved upwards. So, it is address has changed. So, that way at execution time also we may we may need to do binding. So, binding is delayed until run time if the process can be moved during it is execution from one memory segment to another memory segment.

So, need hardware support for address maps like a base and limit registers. So, they basically if it is the base and limit register, so if the address if the program generated addresses are starting from compiler generated addresses are starting from 0, then you see the by modifying this base and limit registers so, you can always do this things. So, you can check whether the program what is the current correct address of the program because by adding that base register value, so you can come to the actual range of memory locations where the process is located. So, this way we can use this binding at various stages and accordingly this process may be developed.

(Refer Slide Time: 08:02)


Multistep Processing of a User Program

So, as far as this; as far as this situation is concerned so, it is like this that the source program, so this compiler it is it is first compiled or assembled that gives us the object module. Then that object module so, it goes through a linkage editor or linker. So, the what this linker does is that it can take a number of object modules and they it can link together into one load module or executable module.

So, basically it may so happen that there may be a big project where a number of group people are working and they are developing their own a module. So, they are writing their source individual source program of different modules. So, as a result so, they come up with different object modules, but ultimately as far as this whole program is concerned so, it has to be put into one load module and that is done by the linker. So, when I have got two different people developing code for the same program same application; so program P 1 and program P 2.

So, program P 1 maybe using some variables which are defined in P 2 and similarly P 2 may define some variable which are defined in P 1. So, if we are if you look into C type of programming maybe here I have got a variable integer x x 1 and P 2 also wants to use it. So, P 2 declares it as an external variable, so telling external integer x 1.

So, as far as P 2 is concerned so, P 2 will compile, but it will understand that these x ones address will come later. And, so, that way all these external variables that we have, so they are marked in the corresponding object modules and when this linkage editor or linker is invoked so, it will be resolving all these external references. So, and that is they are resolved and this linkage editor will generate the load module.

And, after this load module what is done is that a number of such a number of this load modules may be combined together and it is also combined with the system library and then it goes to the loader like there are common routines like for this input output operations reading from keyboard that displaying onto the screen and all. So, this basic input output operations for example, they are part of all the modules. So, it is not required for the users to write that code separately and they are made available from the system itself, so that is attached to all other thing.

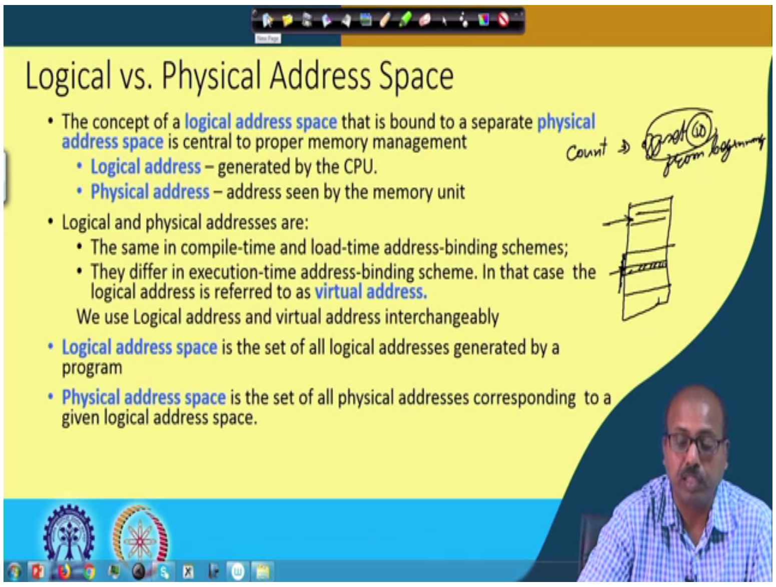
So, apart from that there may be other libraries like math library and all. So, which are basically the all library routines so, they are not written individually by the users every time. So, they are made available by the system time in the form of system library. So, they are also executable file. So, they are made part of this load module and then this loader will load it into this in memory binary ah memory image. So, basically this loader what it does is that this system library plus this load module so, these are taken together and then all of them are loaded by this loader into this thing binary memory image.

And, then you see that apart from this system library there are certain libraries which are dynamically loaded system libraries. So, some system libraries they are loaded statically, some of them are loaded dynamically. So, statically those memory those library routines for which the addresses are known, but dynamically what can happen is that there may be many routines that are there which are they are not made part of the system, part of this module because that may not be required by other programs also. So, as a result they are not part.

But, when the this particular program is loaded into main memory at that time it finds that there are some modules needed which are already loaded in the memory by some other program. So, that way it may like to use those modules, so this loader will determine all those all those library routines and they will be linked up dynamically. So, these dynamically loaded system libraries are they will also become part and this dynamic linking so, it will take care of that situation. So, as a result we get this execution time image of the whole file and it executes in that fashion.

So, this is the multi stage processing of a user program starting from the source program to the in memory binary image file. So, this loader when it is loading the program into memory so, at that time it has to it has to look into the memory space that are available for loading that program and it has to do it accordingly.

(Refer Slide Time: 13:01)



Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - Logical address – generated by the CPU.
  - Physical address – address seen by the memory unit
- Logical and physical addresses are:
  - The same in compile-time and load-time address-binding schemes;
  - They differ in execution-time address-binding scheme. In that case the logical address is referred to as virtual address.

  We use Logical address and virtual address interchangeably
- Logical address space is the set of all logical addresses generated by a program
- Physical address space is the set of all physical addresses corresponding to a given logical address space.

Next, we look into another very important concept known as logical versus physical address space. So, the concept of a logical address space that is bound to a separate physical address space is central to the proper memory management. So, as we said that logical address. So, that is basically the address generated by a program and when it is when the program is generating some address, so even if it is with respect to the beginning of the module so, it is telling that ok, so many offset after the from the beginning of the program. So, like that it is generating the address.

So, these are called logical address and this logical address is generated by the CPU and then this CPU this logical address has to be converted into some physical address because logical address is telling me that you have to you have to take you have to access the variable count which is at offset 10 from beginning or offset 10 from beginning, but memory will not understand that. So, memory will understand only in terms of address actual addresses.

So, if you give this 10 to the memory then what will happen is that it will try to access the memory location 10. But, that is not correct because this particular program may be loaded in this part of the memory and when I say this is the location that I am talking about. So, this is the location that I am trying to access. So, this is the logical address. So, the offset 10 from beginning so, this 10 is the logical address that has to be converted into some physical address before it is given to the main memory. So, this physical address is the address seen by memory unit and logical address is the address generated by the CPU.

So, logical and physical addresses are same in compile time and load time address binding schemes. So, if we are doing compile time and load time address binding; so compile time means, so we have got the addresses are known. So, this there is no distinction between logical address and physical address. Similarly, the load time address binding scheme on the program is loaded then the address are known.

So, there also this logical and physical addresses does not have any meaning. But, this execution time address bindings that is when the program is being executed then this addresses may change ok. So, in that case this logical address is referred to as virtual address and then this after that the virtual address will be converted to physical address.

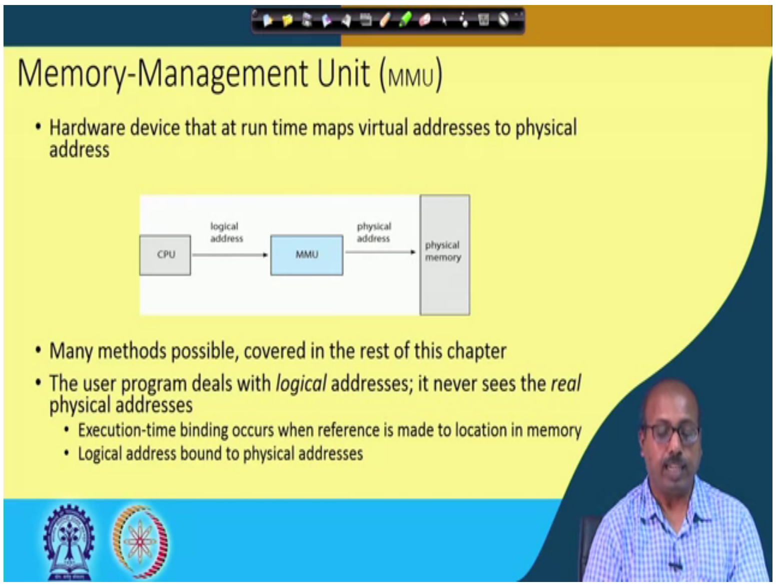
So, in our discussion we will be using this that the term logical address and virtual address interchangeably. So, understanding the thing that this logical or virtual address is the address generated by the CPU and the physical address is the address that is seen by the memory manner by the memory module.

So, logical address space of a process is the set of all logical addresses generated by a program. So, in a in the in a program so, it can refer to all the variables that it has it can refer to the entire portion of program code that is there with the process. So, that will determine the logical address space for the process; whereas, the physical address space is the set of all physical addresses corresponding to a given logical address space.

Now, the program is loaded from different portions in the from a particular portion of the physical memory. So, when the program is there so, it is accessing actually that part of the physical memory. So, that is the physical address space and logical address space is the logical address generated by the program. So, this is the difference between logical space logical address space and physical address space. So, logical address space is generated by CPU and physical address space is whatever they seen by the memory management module.

So, there is a conversion from this logical address space to physical address space and the memory management unit is supposed to do this particular job. So, if it is not done the memory management unit is not present, then we have to with the user can always do it, but as we have seen previously for executing a single instruction, so four more instructions are needed to do this checks and all, so that becomes a costly operation. So, most of the processors that we have, so they have got this memory management module which will be doing this logical to physical address space conversion.

(Refer Slide Time: 17:22)



Memory-Management Unit (MMU)

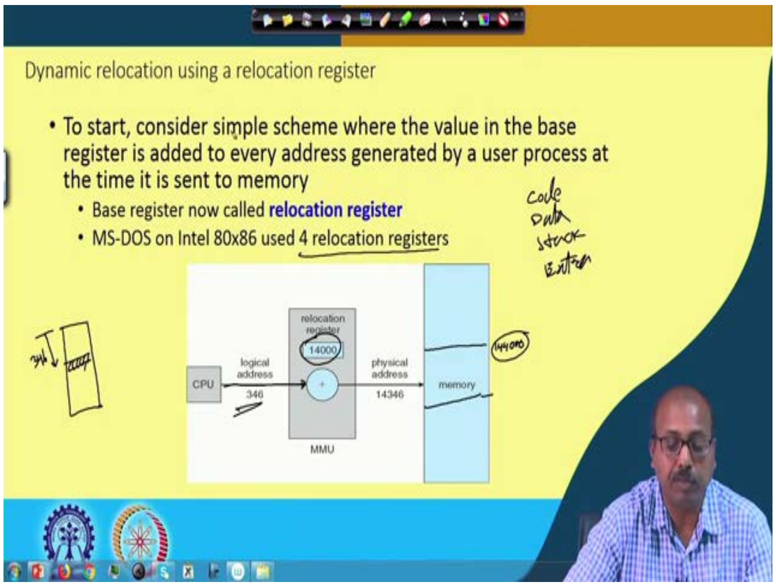
- Hardware device that at run time maps virtual addresses to physical address
- Many methods possible, covered in the rest of this chapter
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

So, as I was telling that this memory management you need so, they are hardware device that at runtime maps virtual addresses to physical address; virtual address and logical address has same. So, this CPU whatever address is generated that comes to the memory management unit and this memory management unit in turn generates the physical address that goes to the physical memory. So, we can have ah different methods of doing this conversion and that each of them will have different level of flexibilities and use of use and all.

So, the many methods are possible, so we will see some of them in this remaining part of this particular chapter. The user program deals with logical addresses and it never sees the real physical address. So, because user program need not be bothered about where the program is actually loaded because if this one also has to be taken care of by the user then the user cannot write the program very easily because it has to ensure that that part whatever address it has chosen so, that address range is available and all. So, and that is very difficult proposition because we do not know when the program is loaded, which part of the memory that will be available.

So, as a result this user program will be talking in terms of logical address space only and it will never see the physical address space, but during execution this physical address space will come. So, execution time binding occurs when the reference is made to location in memory and logical address is bound to physical address at that time. So, this has this process has to be automated this logical to physical address binding, this process has to be automated and that is done by the memory management unit.

(Refer Slide Time: 19:13)



Dynamic relocation using a relocation register

- To start, consider simple scheme where the value in the base register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers

So, this dynamic relocation that is previously the program was there at some accessing some location and how it is going to change. So, this dynamically location using a relocation register, so this is a very simple type of memory management policy. To start, consider a simple scheme where the value in the base register is added to every address generated by a user process at the time it is sent to memory.

So, as I said that at compile time it is assumed that the programs are program module are starting from address 0. Now, if the so, whenever we have got this thing so, the CPU generates an address 346 what it essentially means is that from the beginning of the memory from the beginning of the module. So, if you come to 346 locations so, this is the location I am referring to.

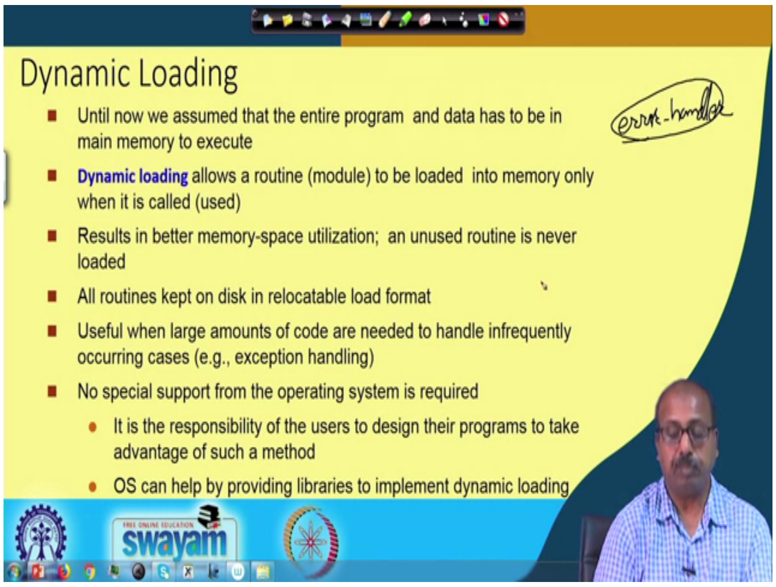
Now, this particular location it is dependent on where exactly the program has been loaded. So, assuming that the program has been loaded from say 144000, so this is the address 144000. So, that is this particular value is loaded into a register called relocation register and whatever logical address is given. So, that is added with this relocation register value so that we get the corresponding physical address. So, 14346 is the physical address.

So, this way if the program is loaded on a different part of the memory then this relocation register has to be loaded separately differently and as a result this program will be the as far as user is concerned so, user does not see that my program is running on a different address space because the if the user sees that it is starting at location 0 only, but in reality it is load it is equal to loaded at different places for each execution. So, base register that we have so, base register is nothing, but the relocation register. So, in this particular scheme, so that is the relocation register.

So, if you look into this MS-DOS operating system on Intel 80 86 type of architecture so, that used 4 such relocation registers. So, using 4 such relocation register so, you can have 4 different addresses for the same memory same program or it may be for 4 different programs we have got 4 different registers. In fact, they can also be used for; in fact, this we have got in a 80 86 program so, there are different segments in the program the code, data, stack and extra segment.

There are 4 segments in an 80 86 program code segment data segment stack segment and extra segment and each of these segments they actually define one portion of; one portion of the user program. So, if so, since there are so each of these segments so, they can be relocated differently. As a result this for relocation registers will be required for relocating these segments independently. So, that is the idea having 4 relocation register in x 86 architecture.

(Refer Slide Time: 22:36)



Dynamic Loading

- Until now we assumed that the entire program and data has to be in main memory to execute
- Dynamic loading allows a routine (module) to be loaded into memory only when it is called (used)
- Results in better memory-space utilization; an unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases (e.g., exception handling)
- No special support from the operating system is required
  - It is the responsibility of the users to design their programs to take advantage of such a method
  - OS can help by providing libraries to implement dynamic loading

error-handler

So, next we will see this dynamic loading mechanism. So, till now we have assumed that the entire program and data has to be main memory has to be in main memory for execution. So, entire program has to be loaded in the main memory. So, dynamic loading so, it allows a routine or a module to be loaded into memory only when it is called it is used. So, far we said that if a program is having say 10000 bytes; so entire 10000 bytes are to be loaded before the program execution starts that by the it is this loading is done by the loader, but this some.

So, that is a bit cumbersome because we require more amount of loading and that 10000 bytes, so that is used reserved by a particular program, the bytes may not be used always. So, it may not be it may be the case that we do not need so many bytes to be loaded at always. So, this dynamic loading mechanism, so it will allow a routine to be loaded into memory only when it is called.

So, results in better memory space utilization and unused routine is never loaded. So, this is the most important part and apparently it seems why there should be some unused routines, but there are unused routines because a program may have some error function or some functionality which is rarely used. So, most of the programs you will find that there are some routines which are called error handler and only if some error occurs then this error-handler will be called. Now, if a program is executing properly then this error-handler will never be invoked and as a result in dynamic loading mechanism, so we do not need to load that error-handler at all.

So, only the portions of memory portions of program which are utilized so, they will be loaded. As a result of the total amount of space required by a particular program will be less. So, it results in better memory utilization and unused routine is never loaded. All routines kept on disk in relocatable load format. So, this is they are kept in relocatable format because wherever the space is free it can be loaded there. And, then if it is required then we will find out a some free space in that in the main memory and put it there.

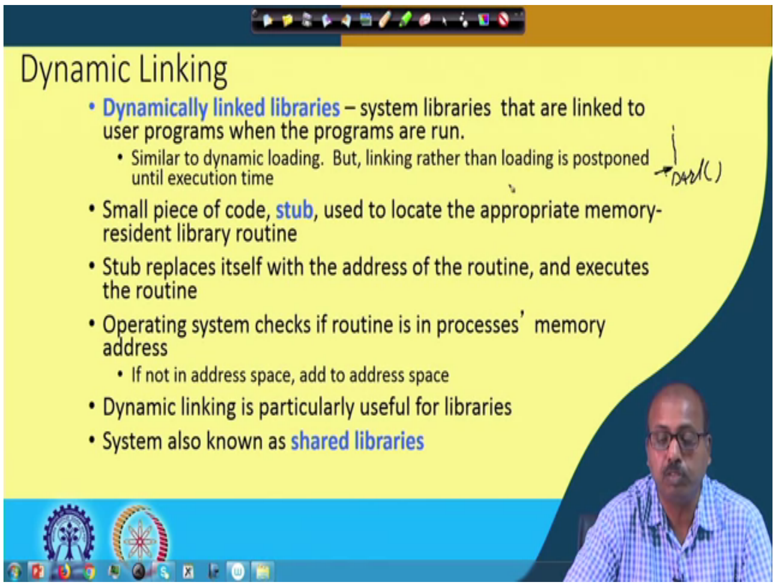
It is useful when large amounts of code are needed to handle infrequently occurring cases like this exception handling. Some exceptional situation like for example, in a particular computer system. So, if a one particular type of input comes then only it we may have to take care of that operation. For example, say fire alarm, so if the fire alarm occurs then only we have to take some action. So, but fire alarm is not going to occur so frequently. So, as a result we do not need to load the routine which takes care of the fire alarm.

No special support from the operating system is required and it is the responsibility of the users to design their programs to take advantage of such a method. So, basically what happens is that the user will understand that this is user will understand that this part is of my program is not going to be loaded frequently.

So, accordingly the program is divided into procedures and such that the entire thing is not a single procedure. It is divided into a number of procedures such that some procedures which are not needed will not be loaded. So, as a result the OS designer they do not have to do anything because the underlying mechanism is like that. So, if it is required then only it will be loaded. So, it they those routines will never be loaded at all.

So, it is the responsibility of the users to design their programs to take advantage of such method. OS can help by providing libraries to implement dynamic loading, so some routines may be provided in the in the form of this dynamic library, shared library. So, that way that can be helpful.

(Refer Slide Time: 26:48)



Dynamic Linking

- **Dynamically linked libraries** – system libraries that are linked to user programs when the programs are run.
  - Similar to dynamic loading. But, linking rather than loading is postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

So, the dynamic linking part; so, dynamically linked library so, system libraries that are linked to user programs when the programs are done. So, when the program, if the library is needed then only it will be linked otherwise it will not be linked. So, it is similar to dynamic loading, but linking rather than loading is postponed than till the execution time. So, maybe my program at some point of time it needs a needs to access one library routine say sqrt, the square root function. Now, until and unless this square root function is called, so this is not going to be linked with the system.

So, if this square root is taken as a static library routine then when the program when the load module is created at that time the linker will be linking this square root routine with the load module and it will be residing as an object file in the, it will be residing as an executable file in the disk. But, but it may in reality this square root function may not be needed ok. So, that is why in dynamically linked libraries so, it is only if the function is called then it will be linked up.

Small piece of code called stub is used to locate the appropriate memory resident library routine. So, in the program code you will find some stubs introduced by the compiler and linker so that actual routine when executing. So, if you need the routine then the stub routine will call the actual routine and that will be that way it will be executed. So, the stub will replace itself with address of the routine and execute that particular routine.

An operating system checks if routine is in processes memory address. If not in address space add to address space. So, it may so happen that this since it is coming as a separate dynamic library, so it may not be within the address space of the process. So, operating system needs to check it. Otherwise it will generate some address error it is not in the address space of the process that way it will generate some address error. So, we have to avoid that. So, this operating system it should tell that this is within the address space of the process.

Dynamic linking is particularly useful for libraries and it is also a system also known as shared library concept ok. So, we have as I was telling that there are several there may be several library routines which are needed by different processes. And, in static library so, they become part of the process; in dynamic library, we have make we put some stubs in the individual processes and only when that particular routine is called then only this library is linked with the processes. The stub processes stub procedure will call that particular procedure and the linking will be done at that time. So, they are called shared libraries.

So, this way the at many times so, we need to do many operations like during execution of the process also we need to do many modification to the process, structure and then all these additional things are to be done. So, until and unless there is a support from the underlying memory management unit, it is difficult to it is difficult to execute this in a by a by users themselves or by a simple user program. And, the user program we loaded a lot for doing all these things. So, the memory management unit so, they are going to help us in all these. So, we will we will continue with this in the next class and discuss on different memory management policies.