

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

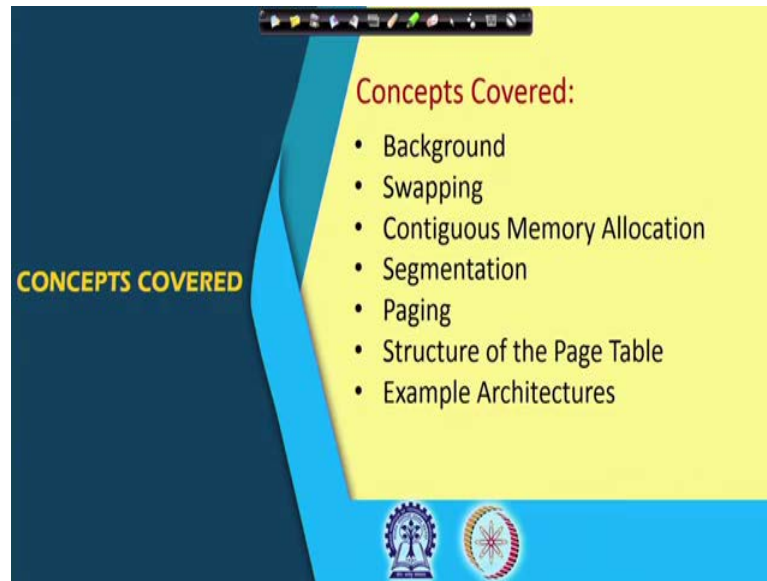
Lecture - 41
Memory Management

So next we will start with the Memory Management schemes that we have in operating system. So memory management is one of the fundamental operations that any operating system will do and memory is another important resource that the computer system has. So, if it is not allocated properly then it may so happen that some process is asking for some memory, but it is not getting it and other process is holding lots of memory, but it is not utilizing.

So we have to be very careful so that we can allow maximum number of processes to continue. At the same time this memory management is difficult because it needs lots of extra calculations that to be done. So extra I should say extra predictions that that are there and we have to do it in such a fashion that the computational overhead does not increase significantly.

So if you look into earlier memory management policy or earlier processor architectures. So, there we did not have any specific memory management policy into implemented as part of the architecture or as part of the processor. But if you look into today's processors the advanced processors then they all have some memory management unit which is built in with the system. So, that helps in the operating system design so that we can go for this faster memory accesses this memory management becomes easy.

(Refer Slide Time: 01:54)



So the concepts that we are going to cover, we will have a background then we will look into a policy called swapping. So swapping is a concept like if the if the memory requirement is high for a process, then we may like to create some additional space for the process by means of removing some of the content of this main memory to the secondary storage and then giving that space to their process. Later on when a process is no more required to be there in the main memory so it is again taken out and put into the disk space. So, that is called the swap out.

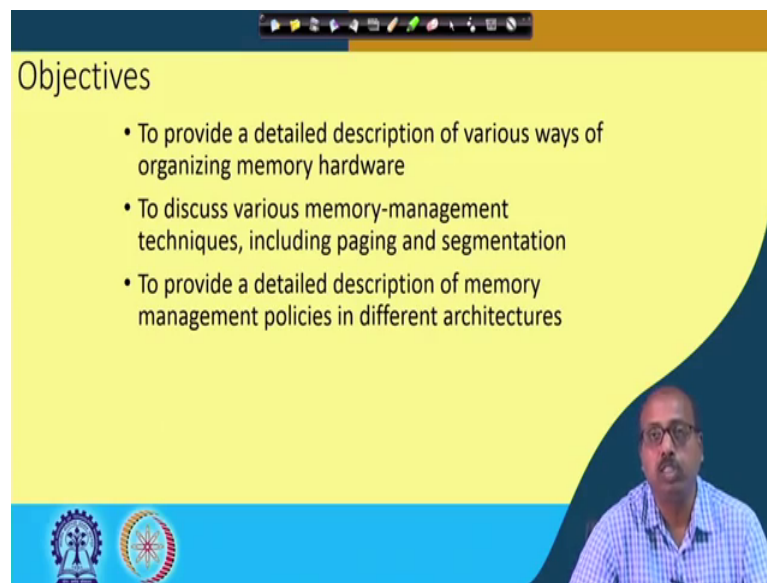
So, it may so happen that during the lifetime of a process initially the process require some of the some portions of its code and data segment. So, those portions are brought into main memory that is called the swap in process. So, later on so, some other parts are needed. So, that these previous parts so, they are returned back to the disk and this new parts are taken into memory. So that is called swap out and swap in process. So, this whole mechanism is known as swapping. So we will look into the details of that.

Then we will look into some memory allocation strategies. The first one is very simple one which is called contiguous memory allocation. Then, we will look into some segmented memory management; so, segmentation scheme, then some paging scheme, then this that then we will have some cost structure of page table and some example architectures. So, these are the major concepts that we have. So, basically this contiguous

memory management, then this segmentation and paging so, they are the three main memory management policies that has evolved over the time.

So, the initial computer systems to they had this contiguous memory allocation, after that came the segmentation and this paging concepts. So, that and after this today you may be familiar that computer systems they have got this virtual memory concepts. So, that we will discuss later in a different class we will discuss about it.

(Refer Slide Time: 04:04)



Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of memory management policies in different architectures

So to start with what are the objectives? So objective is to provide a detailed description of various ways of organizing memory hardware. We discuss various memory management techniques including paging and segmentation and to provide a detailed description of memory management policies in different architectures.

So, these are the three things that we are going to do in this part of the lecture.

(Refer Slide Time: 04:24)

Background

- A program must be brought (from disk) into memory and placed within a process for it to be run
- A program can be written in machine language, assembly language, or high-level language.
- Main memory and registers are the only storage entities that a CPU can access directly
- The CPU fetches instructions from main memory according to the value of the program counter.
- Typical instruction execution cycle – fetch instruction from memory, decode the instruction, operand fetch, possible storage of result in memory.

So a program must be brought into memory and placed within a process for it to be run. So, this is the fundamental thing because the CPU can talk only to the main memory. So, if you from our school days we know that with architecture that a CPU follows is that this is the processor and this processor has got the memory.

So, this processor will get interface with the memory the address bus data bus and control bus. So that the processor will put the address on the address bus and memory will be retained in the content and that statement is executed by the processor and this control bus is giving the control signals like read, write etcetera. So, that is the standard model. So, this processor cannot talk to the secondary storage like disk directly. So, for any program to be executed it must be brought from the disk and into the memory and placed within a process for it to be run. So that is the fundamental requirement.

A program can be written in machine language assembly language or high level language. So, this is the thing so, you can directly write in machine language, you can write in assembly or you can write in high level language. Main memory and registers are the only storage entities that a CPU can access directly. As I said that as I said that CPU cannot access disk directly. So, as far as the storage is concerned so, there is a small amount of storage in terms of the CPU registers that are there in the system. So, that is one thing.

So, the CPU registers and the other one is the main memory that is there. So, in between we have got cache and all but for all practical purposes so, you can take cache to be a part of the main memory because as far as the access is concerned so, what the processor does whenever it need some data so it issues a request to both cache and memory. So, cache may or may not have that content available in it. So, if the cache data if the content is available in cache then the response will come faster, but otherwise it will come from memory.

But that way it does not make any difference like it is not that this processor just makes a request to cache, it does not make a request to memory. So, that is not the situation. It always makes parallel request to cache and memory. So, that way it is so, memory is actually the storage that we have and there are CPU registers, there are certain instructions that a processor has, so that they actually uses this CPU registers as the operand. So, we have got this main memory and CPU registers as the storage entities.

The CPU fetches instructions from main memory according to the value of program counter. So, this is known from our architecture classes. So, we have got this program counter available in the processor. So, that content is put on to the address bus to tell what is the address of the next instruction to be brought next memory location from where the access has to be made. So, CPU features instructions from main memory according to the value of the program counter.

And, typical instruction execution cycle is like this fetch instruction from memory, decode instruction, and then after decoding the instruction it may so happen that the operands are there in the memory. So in that case it has to do an operand fetch from the memory and then it is; it will do the operation. So, if operand fetch is not from the memory so the operands are in the CPU registers themselves.

So, whatever it is the operand fetch maybe from the CPU register, maybe from the main memory and after fetching the operand so it will do the operation and then the result will be stored in the main memory. So, that is actually the sequence of operation that takes place in an instruction execution.

So what I want to mean is that this memory access becomes a very vital one. So, while instructions are being executed, and since this memory access is off chip so, if you can do it; if you can do it efficient in terms of access time and all then only the system

performance will be good. And this memory physical memory size so, it puts a limitation like a program maybe very big in size and if it is not fitting into the main memory, then we cannot have that. So, this we cannot run that program in the system.

Of course, with the concept of virtual memory and all so, this limitation has been pushed off significantly. But, still for a very simplistic viewpoint you can say that for a program to execute it must fit in the main memory. So this main memory size is an important issue. So, we will see that how this memory management policy so that will be affected by this size of the memory and all.

(Refer Slide Time: 09:23)

Background (Cont.)

- Memory unit only sees a stream of one of the following:
 - address + read requests (e.g., load memory location 20010 into register number 8)
 - address + data and write requests (e.g., store content of register 6 into memory location 1090)
- Memory unit does not know how these addresses were generated
- Register access can be done in one CPU clock (or less)

The diagram shows a memory unit with an address bus (A) and a data bus (D). The address bus is connected to a register (R) and the data bus is connected to a memory unit (M). The memory unit has read-write control signals (RR).

So, memory unit only sees a stream of one of the following address plus data requests. For example, load memory location 20010 into register number are 8 so, this may be one operation that has to be done. And, address plus data and write requests, so, this is like store the content of register 6 into memory location 1090. So, that way either load or store instructions may be coming so, to access a memory location for a read operation or for a write operation. So, this read and writes operations are there, so, they have to be done.

So, memory unit does not know how these addresses were generated. So, on the address bus the lines are coming so, any memory chip if you see then this memory chip has got this address bus. So it has got the address bus, then it has got the data bus, you have got the data bus and there is read-write control. So, this is the typical situation. So, we are

depending upon the address that is put on to the address bus so and the control signal given read or write so, it will access the particular location and perform either a read operation or a write operation.

So, memory is not bothered like how these particular address data so, these were generated. So, the how did it come to here? So, whether it is by some processor or by some other module etcetera. So, that is not known. So, that is what the memory unit does not know how these addresses were generated. Register access can be done in one CPU clock because register is within the chip so, it can be within the CPU chip. So, it can be done very fast. However, maybe even less than a clock also maybe the access may be done.

However, for memory access so, it takes more time because it has to go off chip. So, as I said that off chip access is taking more time. So, it may require more time. So, it is advisable that we have got many of the operands loaded into the CPU registers so that this access is faster.

(Refer Slide Time: 11:35)

Background (Cont.)

- Completing a memory access may take many cycles of the CPU clock. In such a case the processor needs to **stall** since it does not have the data required to complete the instruction it is executing.
- **Cache** sits between main memory and CPU registers to deal with the “stall” issue.
- Protection of memory is required to ensure correct operation:
 - User process cannot access OS memory
 - One user process cannot access the memory of another user process.

Completing a memory access may take many cycles of the CPU's clock. So, because memory is off chip, so it will take quite some time. In some case; in such a case the processor needs to stall since it does not have the data required to complete the instruction it is executing. Maybe it is doing say $A = B + C$ and out of that this B and C so, they are to come from the memory. So as long as this B and C are not

available from the memory so, this processor has to wait. So, processor is in some sense the processor is stalled. So, it is waiting there.

So that is the time required is maybe pretty high because it has to access this memory the processor has to access the memory. So we can many systems we have got a cache which is sitting in between the main memory and CPU registers to deal with the stall issue. So, what is done is that if this is the CPU and this is the memory, then we have got some cache which is there are several types of cache level 1 cache, level 2 cache etcetera. So, level 1 cache is within the CPU level 2 cache is outside the CPU. So, this is L 2 cache. So, this is called L 1 cache. So, like that.

So, whatever it is, so, this if the cache is there then when CPU is asking for some data. So, it is putting the address on the address line so, that comes to the cache as well. So, it goes to the cache as well as it goes to the memory. Now, cache is designed using a different technology called content addressability so that it will access all these locations simultaneously to see whether this particular address is available in this cache or not.

So, if it is available in that case it will respond to the CPU that ok, this is the with the corresponding data. So, that access because of this mechanism that this cache memory has so, it is much faster compared to memory or to compare to main memory. And, if it is on-chip then it walks almost at the same speed as the CPU.

So, that is why cache access is going to be much faster and it helps in thus is improving system performance. However, the difficulty is that the cache is not cheap. So, it takes quite it makes the system costly. So, you cannot have a large cache. So, only some small portion of this main memory can be made a part of cache. And, what is done is that whenever a CPU requests a particular address and that address is not found in the cache so, if it is found in the, it will definitely be found in the memory. So, from memory some sufficient number of bytes which are neighbor to that so, they will be loaded into the cache.

So, that in near future the when the CPU requests for those locations so, they are already available in the cache. So, that way there is some updation that goes on. So, the old data that is there in the cache which is no more required by for program execution. So, they get overwritten by this newly required content. So, whatever we do so, this particular policy so, what it ensures is that this cache sitting between the CPU and main memory so

it can reduce the memory access time. So, that is in some senses it can deal with the stall issue.

Protection of memory is required to ensure correct operation. So, protection is very important because it may so happen that as far as this processor execution is concerned you see so, this entire memory appears to be similar. So, whatever you want to do a read-write operation so, it will be doing a read-write operation there.

But, in many system; in any computer system a part of the main memory, so that is reserved for the operating system. So, important content like say operating system code, then important data tables and all like process table, PCBs, then your file table and all. So, they are now important OS data structures. So, they are also made in the main memory.

Now, if a program by mistake or intentionally tries to modify the content of those OS locations so, to corrupt the OS. So, that way, that has to be prevented. So, these user processes it should not be able to access this OS memory and one user process cannot access memory of another user process. So, it may so happen that I have got a multi user system where this memory is divided into some regions and maybe this part is given to user 1, this part is given to user 2 like that.

So, process of user 1 should not be able to access the data and stack of process in of user 2 and vice versa. So, user 2's process would not be able to access the data elements of user 1 process. So, that way one user process cannot access the memory of another user process that also has to be protected. And, as I said that as far as programming execution is concerned CPU cannot make this differentiation between which process data it is.

So, as a result we have to do something extra on top of the general model that is the user if the processor putting the address on the address bus and all. So, over and above that particular model we have to do something so that this checks are made and this protection is ensured that is one process does not access data of another process and the process does not access or modify, the operating system data. So, these are the things to be done so, we will see how these are done in different memory management policies in our successive lectures.

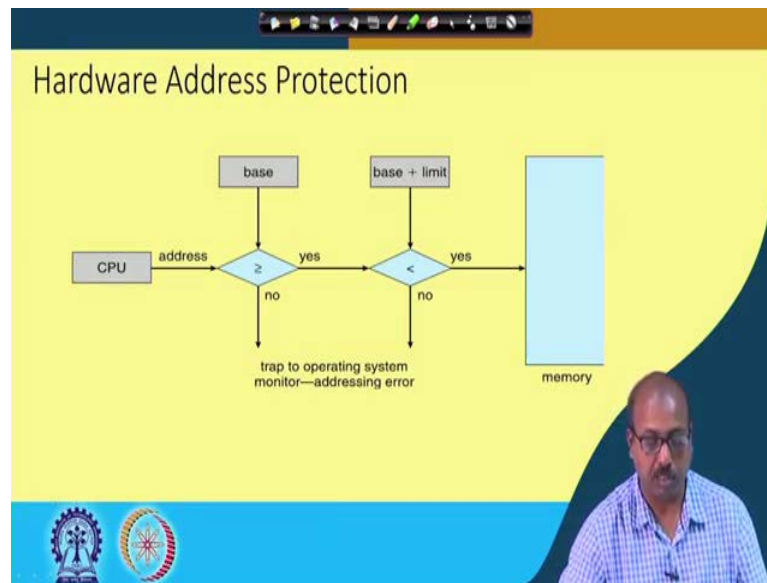
And, CPU must check that every memory access generated in user mode is between the base and base plus limit for that particular user. So, that has to be insured by the CPU. Now, if you see that every so, so any memory access now is now converted to a number of checks a number of additional instructions to be executed. So, if you are not having any help from the underlying hardware then for any memory access say, for example, if I have got a instruction load some address XXXX then before that the CPU must check whether this XXXX is valid within this limit or not.

So, the CPU should compare this base of the process with XXXX, jump on less or equal to invalid because it is not there then it will compare the limit plus base; it will compare this limit plus base with XXXX. And now this value should be so, this fellow should be this XXXX should be less than this limit plus base. So, this jump on greater to again invalid otherwise only it should have this load XXXX instruction.

Now, you see that a single instruction in during execution so, it has to be converted into this five instruction. So, that way it is becoming costly to ensure that protection. So, if the underlying hardware does not provide me a provide me any support and if I want to do everything in software, then you see it is taking low good amount of time for doing this check and all. So, CPU but this check is mandatory because every address that the CPU generates we need to check whether it is within the limit of the current users memory space. So, that has to be done.

So, this memory management unit will help us in this process. So, that will have all these limit register, base register etcetera and that will have the automated hardware. So, that will whenever you execute a load type of instruction so, it will check the limits with this value that value of the address. So, that will be done.

(Refer Slide Time: 22:28)

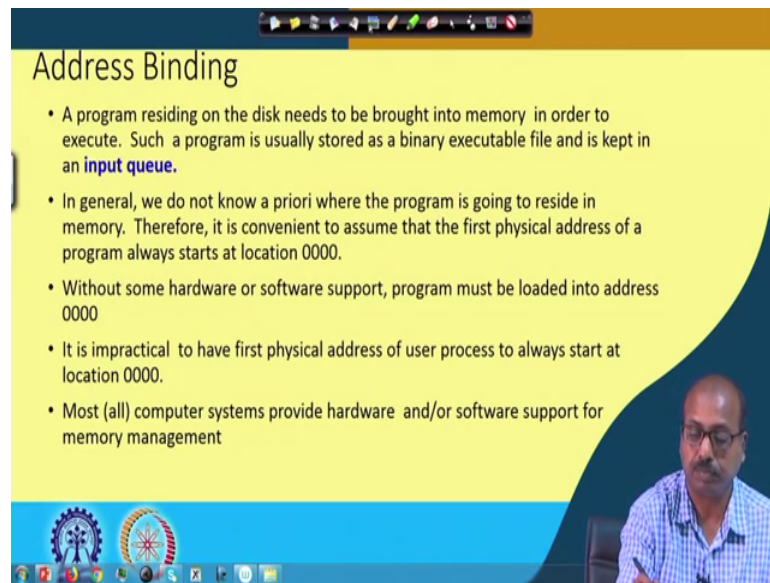


So, this is the hardware address protection. So, CPU generates an address. So, we check whether the address is greater than base or not. So, if it is no, then it is a trap to operating system monitor that addressing error. So, there is a so, if the address is greater or equal base then it is coming through this yes line.

So, it is reaching to the next check where it is comparing this address with base plus limit. Now, in this case if it is less than the address is not less than base plus limit so, in that case this is again a trap to the operating system that there is a addressing error. If it is yes, then only it will be allowed to access the memory.

So, you see that this is the additional checks that I was talking about. So, any address that is generated so, it has to go through two additional checks to see that address is valid and address is within the address space of a particular process and that makes it costly. So, we will see how this can be resolved like many of the processors they have this thing.

(Refer Slide Time: 23:32)



Address Binding

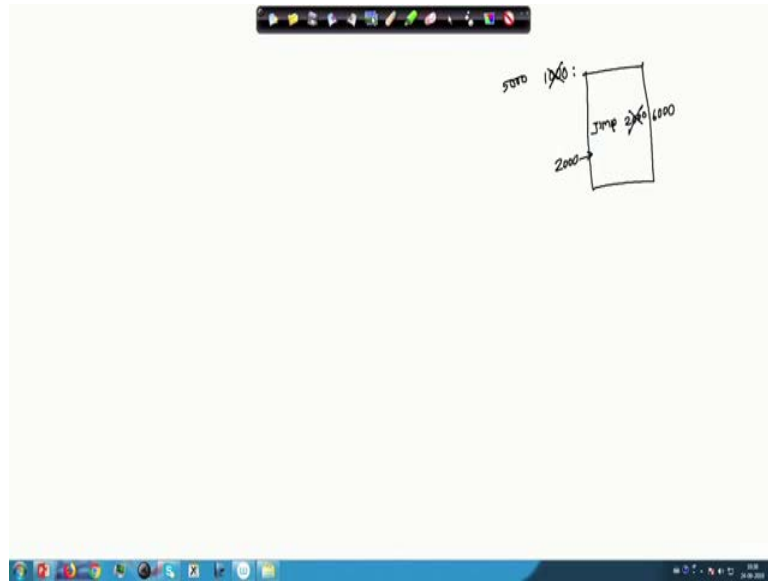
- A program residing on the disk needs to be brought into memory in order to execute. Such a program is usually stored as a binary executable file and is kept in an **input queue**.
- In general, we do not know a priori where the program is going to reside in memory. Therefore, it is convenient to assume that the first physical address of a program always starts at location 0000.
- Without some hardware or software support, program must be loaded into address 0000
- It is impractical to have first physical address of user process to always start at location 0000.
- Most (all) computer systems provide hardware and/or software support for memory management

The slide features a yellow background with a dark blue header and footer. A small video inset in the bottom right corner shows a man with glasses and a mustache, wearing a blue and white checkered shirt, speaking. The footer contains several logos, including the Indian Institute of Technology (IIT) logo and a circular logo with a gear and a person.

Next issue that we have is that of address binding. Now, the point is when a program is there in the disk so, when the program is ready for execution the user ask the computer to execute that program, so, the program has to be loaded into the main memory. Now, in a multi user system it is very difficult to predict like what is the address at which the program will be loaded.

So, that makes it because once the program is run maybe the program is loaded from location 1000, next time the program is run it is loaded from location 5000. So, the difficulty that comes is like this.

(Refer Slide Time: 24:18)



Suppose, I have got a program so, if it starts at location 1000 then I have got at some point of time I have got a jump to location say 2000 or 2000 is somewhere here. Now next time the program is loaded so, instead of being loaded from 1000, it is loaded from say 5000. Now, this jump 2000 is no more valid so, it has to be jump 6000. So, that way when a program is staying in the disk so, it is not known like where the program is going to be loaded and somehow we have to resolve this issue. So, this address binding is actually trying to answer this question. We are trying to see like when the where the program will be loading.

So, program deciding on the disk needs to be brought into main into memory in order to execute such a program is stored usually in a binary executable file and kept in an input queue. So, as the program is there is a binary executable file and that is there in the disk and it may be in the form it is kept in a queue that it will be loaded from the secondary storage to the main memory.

In general, we do not know a priori like where the program is going to reside in the memories. So, as I was telling so, depending upon the availability of the memory space the operating system loader it may decide to load the program from different address next time it is loaded. So, therefore, so as far as the code generation is concerned the compilation or assembly is concerned so, it does not know from which address it will the program will be loaded. So, what it does, they those tools they will conveniently assume

the first physical address of a program to be location 0, 0000. So, that is the location 0. So, it assumes that the program starts from location 0.

Without some hardware or software support program must be loaded into address 0 because if it is not so, then all those address sensitive instructions. So, they will start misbehaving. So, the program must be loaded from start from address 0. However, it is impractical to have the first physical address of a process to always start at location 0, because in a multi user system we have we may have other processes running.

So, we may not have the say the address 0 available and in most of the operating system the initial address locations so, they are reserved for the operating system. So, this location 0 etcetera so, they are always reserved for the operating system.

Most or all computer systems they provide a hardware and a software support from memory management. So, this has become almost evident now that the any computer system or any processor design so, they will have some mechanism for this for providing this hardware support for memory management. So, this has to be done.

(Refer Slide Time: 27:18)

Address Binding (Cont.)

- In general, addresses are represented in different ways at different stages of a program's life
 - Addresses in the source program are generally symbolic
 - For example, variable "count"
 - A compiler typically **binds** these symbolic addresses to relocatable addresses
 - For example, "14 bytes from beginning of this module"
 - Linker or loader will bind relocatable addresses to absolute (physical) addresses
 - For example, 74014 ~~74014~~
 - Each binding maps one address space to another address space

So, we can see the how this is going to take place. So, in general addresses are represented in different ways at different stages of a program's life. So, programs life means that initially the program is written in a text file, then it is compiled in an object

file and the and finally, the program is loaded on to the memory for execution. So, these are the different stages of life of a program.

So, in general addresses are represented in different ways at different stages of program of a programs life. Address addresses in the source program are generally symbolic. For example, we have got the variables neither we represent them by means of their names like maybe the we have got a variable called count, so, in my program I am referring to it as a count. But, when the program will finally, be loaded this count will not have any meaning. So, count will basically be a memory location so that address of that will become the important one. So, they are the name count does not have any beery.

A compiler typically binds these symbolic addresses to relocatable addresses. So, what happens is that, suppose I have got a in my program I have got an integer variable count. So, what the compiler does is that so, compiler assumes that the code is generated from address 0. So, the there may be other variables defined here. So, as a result maybe the count is coming at this point.

So, what is the offset that is starting from the beginning of the program after how much; after how many bytes this variable count will come into existence? So, it may be 14 bytes from the beginning of the module. So, that way so, this is the binding so, compiler will bind this count variable to be offset 14 from the beginning of the module.

Then after that the linker or loader will bind relocatable addresses to absolute addresses. So, this is called a relocatable address because it does not matter like from where the program is loaded whether it is from 0 or whether it is located from located from 5000. So, if I say the I am talking about the memory location which is 14 bytes away from the beginning of the module. So, if I say like that so, it does not matter where exactly my module is loaded. So, that is why it is called a relocatable address.

So, a linker or loader will bind this relocatable addresses to absolute or physical addresses. For example, it may so happen that my program starts from the location 74000 and I said that it is 14 bytes from the beginning so, plus 14. So, the address of this count becomes equal to 74014. So, this way this linker or loader so it will modify this reloadable address into absolute address or physical address and then each binding it maps one address space to another address space.

So, if for example, initially from this textual description so, it is a mapping on to something like this that is the distance from the beginning of the module. And after that when this program is loaded into main memory so it is giving me a offset from this relocatable address to the physical address. So, that way that is another level of mapping. So, these bindings step it will be mapping one address space to another address space.

So, this is the basic idea how this programs are loaded into memory and then how these addresses are resolved. So, this will help us in loading the program at different places. So, from this point onwards so, we will assume that programs that we loaded anywhere in the memory depending on the availability and these issues of these addresses and also they can be resolved easily by the linker or loader.

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