High Performance Computing

Prof. Matthew Jacob

Department of Computer Science & Automation

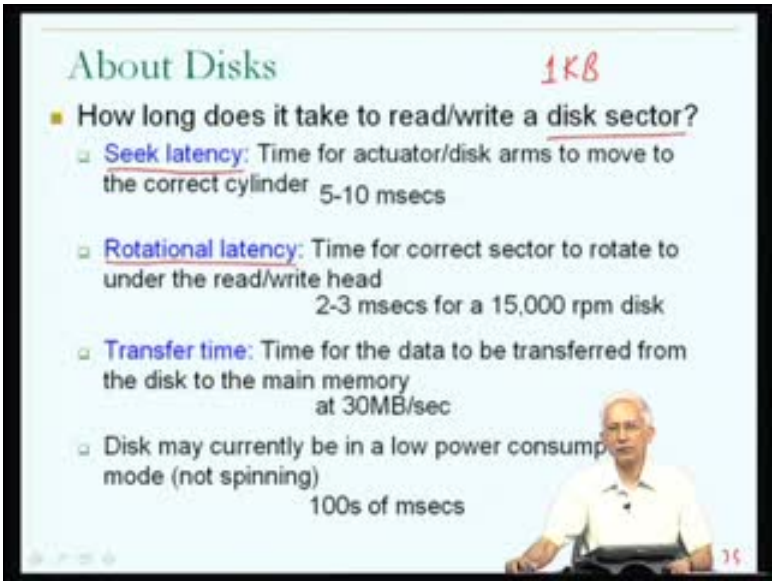
Indian Institute of Science, Bangalore

Module No. #08

Lecture No. #35

Welcome to lecture thirty-five of the course on High Performance Computing. We are currently looking at topic which talks about file systems. File systems are the mechanism through which we are able to have data of a life time which can continue to exist beyond the execution time of a program. And we have realized that, in order to do this, the data of a file will have to be stored on what is called a persistent or a non-volatile storage device such as a secondary storage device, the example of which we are using is the hard disk.

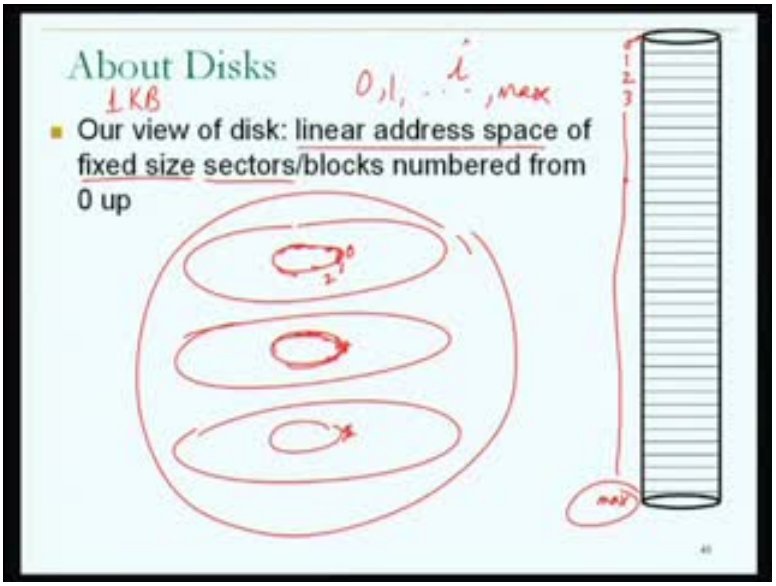
(Refer Slide Time: 00:47)



In the previous lecture, we had looked at typical structure of a hard disk today and had understood something about the amount of time that it takes to read a disk sector. A disk sector is the basic unit of read or write from a hard disk and the size of a disk sector could be something like 1 kilobyte. We saw that there are two main components to the

amount of time that it takes to access a disk sector from a hard disk- one is called the seek latency; it relates to the movement of the disk read/write arm in or out, the second is the rotational latency it relates to the rotation of the disk platter until the correct disk sector comes under the read/write head.

(Refer Slide Time: 01:34)



Now, with that complicated understanding of how a disk actually operates, we will go back to a simpler model of what a disk looks like, from the perspective of managing it from the operating system side. Now, when we looked at main memory, we used a very simple model of main memory where there was a linear address space; in other words, there were, main memory was made up of bytes, the each main memory location had to have an address; otherwise it could not be referred to.

And the lowest address was consider 0 and the depending on the size of main memory, you went from 0 to 1 up to sum maximum possible address. And there were some benefit of viewing main memory in that way, in that, it give us a symbol interface for doing things with main memory such as virtual address spaces and so on. We understood later that the situation was actually much more complicated. A program may view memory as being from virtual address 0 up to virtual address 2 to the power n minus 1, but in reality the different components of that virtual address space could be in different places on disk or in main memory at adequate particular point in time.

So, will have similar, simple module as far as disks are concerned. I will continue at this from this point on, we will view the disk as being a linear address space of fixed size sectors or blocks and will number the sectors from 0 up. So, very simple, similar model to what we had for main memory. So, the key things we note here are that, we are going to assume a linear address space. In other words, we are going to have disk sectors or blocks which have addresses which were going to number from 0 up to some maximum possible value depending on the capacity of the hard disk.

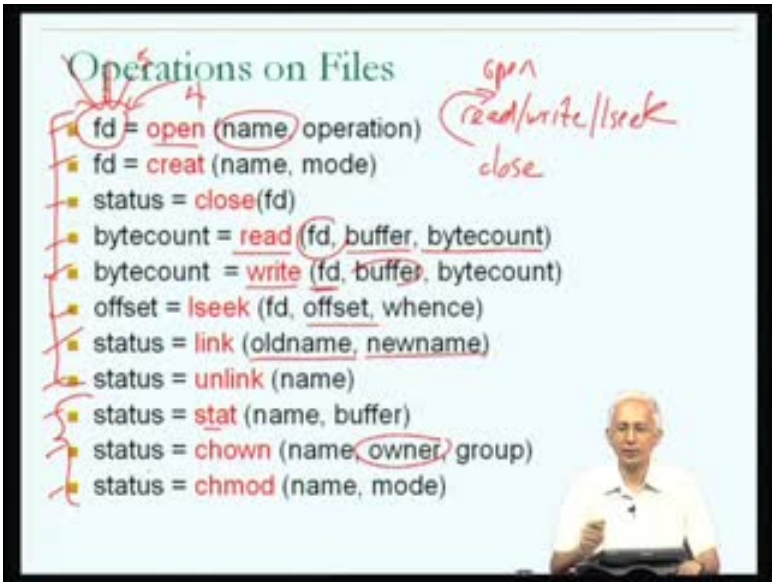
We know that each sector is of some fixed size just like cache blocks were of fixed size, just like pages were of fixed size, we know that the sectors are of a fixed size and approximately, our understanding is that they may be depending on the hard disk. The disk sector may be of size 1 kilobyte or there abouts. So, this is the model that we are going to have of a hard disk and therefore, at this point, if I had to draw picture of a hard disk, I might draw something that looks like this. It looks very much like our picture of main memory. So, there is disk sector 0, there is disk sector 1, there is disk sector 2 and this keeps on going until, depending on the size of the hard disk, some maximum disk sector address.

And we, in the certain cases, it may be necessary like for somebody who is building an operating system to try to relate this linear address space of disk sectors to what is actually happening in the case of the disk. If you think about it in the case of the disk, we know that the disk is made up of many platters and that a least platter, there are many tracks. And there on each tracks, there are many sectors. And now if I talk about disk sector 0 1 2 and 3 in this linear location, there are possibilities that when I looked at the actual disk, I may find out that disk sector 0, from this diagram, is over here and disk sector 1 might be over here and disk sector 2 might be over here. On the other hand, it is possible that it is not the way that the linear address space is mapped, but the disk sector 0 is followed by disk sector 1 is followed by disk sector 2 and so on on that same track, and that beyond a certain number of disk sectors, we may move to the another track of the same cylinder and so on.

So, this is in level of complication that we are not going to get into. We will just work with this linear picture of how disk could be accessed. We know that there are a large number of sectors available on this disk; the number of sectors is going to be determined by the number of surfaces multiplied by the number of tracks per surface multiplied by

the number of sectors per track. That will tell us the total number of sectors that are available on a hard disk. But we will work with this simple linear address space model for the rest of our discussion. So, as far as we are concerned, we could talk about disk sector i, where i is some unsigned integer between 0 and max.

(Refer Slide Time: 05:34)



Now, we have already seen that the operating system manages the file systems which is why we know that there are system calls associated with file systems in most of the operating systems that we are familiar with. And many of the system calls we have seen before, when I talked about system calls, I talked about the file system calls such as create; which could be used to create a new file. Now, the mode of operation, if you have a program that it is going to use a file and the file already exists, then one has to start by opening the file. In other words, informing the operating system that this particular program is going to use this particular file and the mode of operation is that, once a file is opened, the operating system returns something called a file descriptor which is a small integer. And subsequently, while opening the file, the user have to provide the name of the file within the program, but subsequently in order to do operations on the file, the user just has to inform the operating system of which file it is referring to, using the file descriptor.

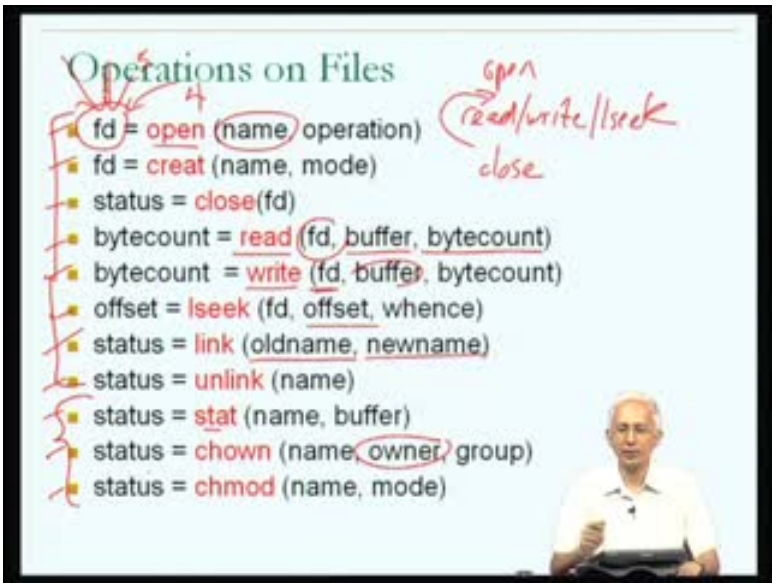
Therefore, it becomes possible to read from a file just by mentioning file descriptor. One can ask for a certain number of bytes to be read from the file into a buffer. The buffer is a

variable of the program. Similarly, one could write from a buffer into a file using its file descriptor. One could move the current byte of interest inside the file to some arbitrary location within the file using lseek and finally, when the program has finished dealing with the file, it could inform the operating system of the same by using the file system call.

So, the typical protocol through which a program uses a file which already exists, is to start by opening the file and then do read or write or lseek operations on the file in order to achieve its objective in terms of using the data- the persistent data or creating more new persistent data, and when all of this activity has been completed, then the program has to close a file. So, there could be multiple reads, writes and lseeks which happen between the open and the close. Further, any one program could have many open files at the same time.

So, I could have a program which is dealing with five different files. It could start by opening all the five files. As a consequence, each of the five files would have a different file descriptor. Remember that, the file descriptor is a small integer. So, the first may have a file descriptor value of 4, the second may have a file descriptor value of 5 and so on. And the program can read or write multiple files by having many of them open at the same time. And those of you have written programs which use files, you will be familiar with this.

(Refer Slide Time: 09:07)

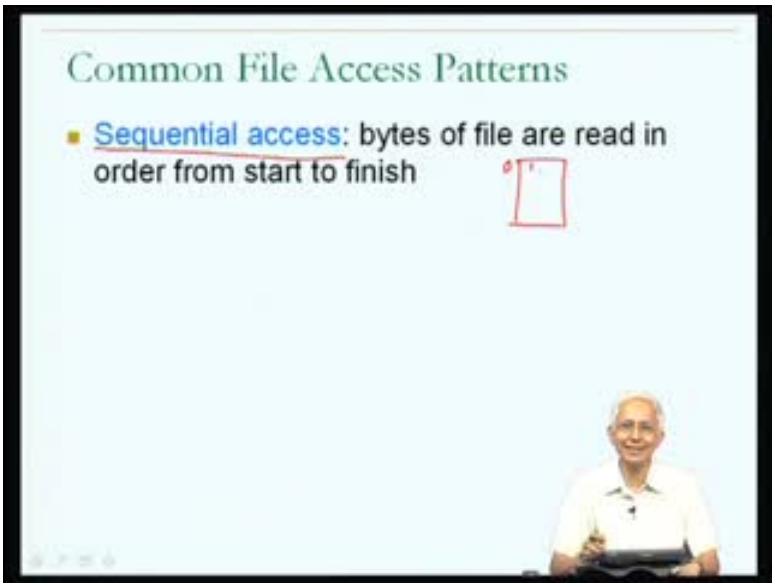


Now, in addition to these basic file system operations, there are others. For example, I had talked about the unlink system call as a system call which could be used to cause a file to seize to exist. By the same token, just since there is a name unlink, it is not surprising that there is a another system called calllink, and this has something to do with creating a new name for an existing file which is why one can think about link as having a property of a new name to be associated with a file which already exists.

Hence something like a situation for a single file, we have two different names by which we could refer to the file. It is in general, the idea of the link operation. In addition to this, there may be system calls through which one could get information about the status of a file and as we learn more about file systems, we could get some understanding of how the status of a file could differ and finally, as we are going to see, there may be the need to manage files at a metal level, possibly by changing the kinds of operations that could be done on a file or by chaining or by changing the operating systems understanding of who the current owner of a file is or the relationship between the owner and other users on the system

So, in addition to the basic set of operations, file system operations that we knew something about from our earlier discussion of system calls, I just wanted to point out that there could be other operations as well. And all of these would be referred to as the file system calls, typically provided by the operating system.

(Refer Slide Time: 09:42)

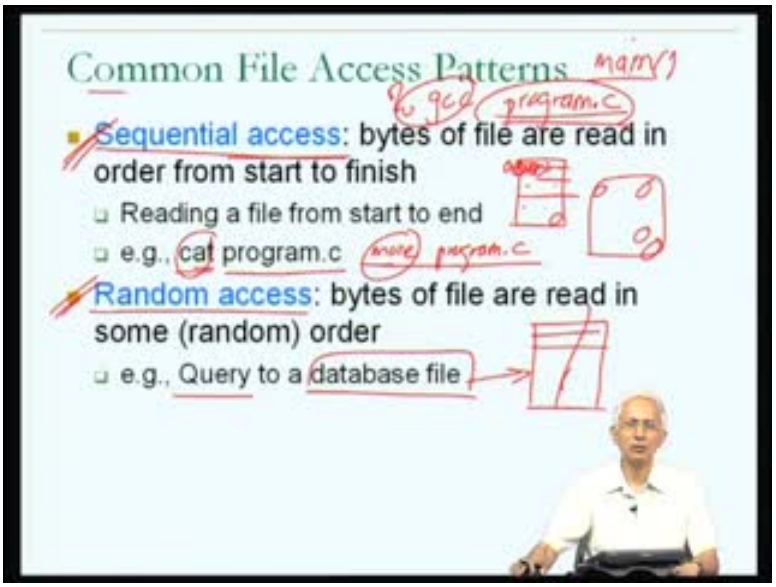


Now, before we get into discussion about how the operating system manages files just as at the time before we got into discussion about how the operating system manages main memory, we had to make some kind of a discussion about typical kinds of program behavior as far as main memory was concerned. And you will recall that it was at that point in time that I introduced the concept of the principle of the locality of reference. The principle of locality of reference was quite useful to us in understanding how operating systems make replacement decisions or operating system policies are decided and basically, the idea was that the operating system uses some modular, some understanding about how programs behave, in order to make decisions at run time.

Similarly, as far as file systems are concerned, there is a simple, similar requirement to understand how operating systems make decisions. How people who write operating systems make decisions as about how to build a file system. We have to understand what are the typical kinds of ways that files are used. In other words, what are the common file accesses patterns. I am going to talk about two common file accesses patterns. Now, one is what is known as sequential access and the idea of sequential access is that the bytes of a file are read in order, starting from the beginning of the file and going on until the end of the file. And this is what is not surprising that is called sequential access- one is sequentially accessing all the bytes of the file starting from the beginning of the file and going on till the end of the file.

Now, just as I could talk about numbering the disk blocks of a disk, I could also talk about numbering the bytes of a file. So, if I have a file, I could talk about the beginning of the file or the very first byte in the file as being byte number 0, the next byte in the file as being byte number 1 and so on, regardless of what the bytes are. I could number the bytes in a file and that is also where the concept of the lseek system call comes in, where when one could move the current point of interest in the file from which reading or writing is being done to some later byte within the file.

(Refer Slide Time: 11:45)



So, the general idea of sequential access is that one is reading a file from start to end. So, one starts by reading the first byte, then one reads to the second byte, then one reads the third byte and so on. Finally, one reads a last byte. And you may ask why do I put this down as a common file access pattern. When I talked about locality of reference, I talked about how instructions and programs have common features like while loops, for loops, array accesses and so on. And we convinced ourselves that locality of reference was a believable property that many programs might show the principle of temporal and spatial locality of reference. We will have to do something similar here. We will have to satisfy ourselves that there are frequently occurring situations where programs may need to access a file from beginning to end. And let me just give you one or two such examples. Now, many of you, when you are on a unix system, may need to look at a program from beginning to end and one way to do this is actually to use facility called cat.

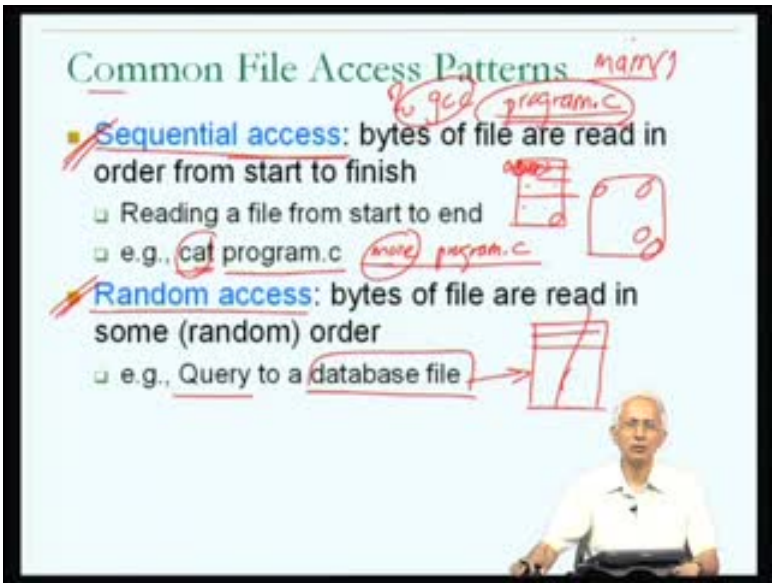
What does cat program dot c do? It basically displays on the screen, the program to you from beginning to end. And if you wanted to get it done on a screen at a time mode, so that you could actually read the file, you might use more program dot c. Now, both cat and more are just programs. And what does the cat program do? It basically opens the file, program dot c, and reads it byte by byte, starting with a first byte and going on byte by byte until it comes to the end of the file. And after reading a byte from the file, it prints it out on to the screen. Similarly more does something very similar, but in the more structured fashion.

Therefore programs like cat and more are; obviously, going through a file sequentially from beginning to end. Now, that is not a very convincing example because the one used cat or more because one wanted to look at the contents of a file from beginning to end and therefore, its fairly obvious that cat or more would refer to the contents of the file and read the contents of the file starting with the first byte and going on till the last byte. But let me give you a more interesting example. Let us consider the situation that, you actually typed in gcc prog dot c, in response to the shell prompt. And you ask the question; What is the file access pattern to prog dot c; as far as this program is concerned. For the moment, we just consider gcc as being a program.

So, gcc is a program which takes as input, the file program dot c. And we are concerned about what is the common file access pattern as far as the way the gcc uses the file program dot c.

Now, one of the first things that gcc has to do is, in fact, to when we talked about gcc, we talked about how it has various steps. And one of the things that gcc has to do is, in fact, to break up, read and understand the contents of the program dot c file. And in order to do this, it really does have to read the contents of the file from beginning to end and this is in fact, another example of sequential access. So, during the compilation of a program, there is a point at which gcc has to read the contents of the file byte by byte from beginning to end, and in the process what it is doing, is something called lexical analysis; that is, it is actually breaking the contents of the file up into the equivalent of words.

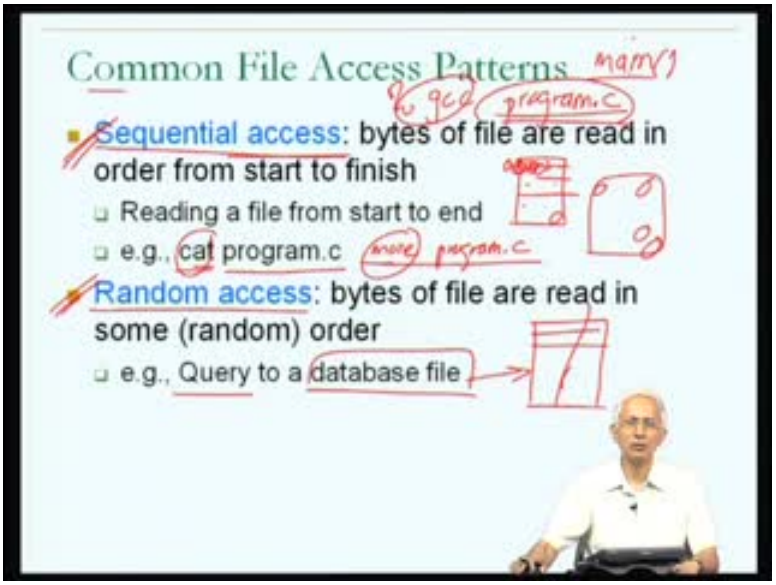
(Refer Slide Time: 11:45)



The contents of the file are in the form of characters, but for example, we know, in the file, m a i n occur as characters. But m a i n has meaning as a word, as far as the program is concerned, the meaning of the program. And therefore, one of the steps in gcc is to actually analyze the lexical structure of the program by reading it byte by byte and breaking it up into these word like chunks.

So, gcc too, in its job will go through a file sequentially from beginning to end and many of the programs that you can think of which deal with data will in fact, have this property that they go through a file sequentially starting from the first byte and going until the last byte. Therefore, with this, it seems that sequential access can be viewed as being a common file access pattern. And it is good to have in mind because it is a very regular kind of a file access pattern. And in operating system might be able to use this regular kind a feature; the idea that a file is read byte by byte from beginning to end in improving some of the policies that it uses.

(Refer Slide Time: 15:54)



Now, another one of the dominant is frequently occurring file access patterns unfortunately is not so friendly. It is what is known as the random access file access pattern and it has the property that the bytes of a file are read in some unknown or random order.

In other words, completely unlike sequential access where we know that it starts with the first byte and then goes on to the second byte and then goes on to the third byte and so on. In the case of random access, the access could start with a byte over here; somewhere deep into the file and then it might read a byte over here, and then it could go to the last byte and then it could read a byte over there. So, there is no order that one could design in the nature of the access pattern as far as the sequence in which the access to the different bytes of the file are taking place. So, this is the other extreme from sequential access and unfortunately is an example of another frequently occurring file access pattern.

Now, you may be wondering what kind of a program that you have dealt with might actually access the contents of a file in some arbitrary kind of a fashion. The answer that I could give you, one answer to this to the doubt relates to a particular kind of file called a database file.

What is a data base file? A file which contains a structured collection of information about a particular data of collection of data is what is referred to as a database file. And there are special programs called database management systems which are used to, by users, to access the information inside a database file.

I could for example, have a database file which has a lot of information about each of the citizens of a country so that, we have a very large database file. And for each citizen, it might have one record of information; the name the address the age etcetera.

Huge number of pieces of information about the first citizen followed by a huge number of pieces of information by the second citizen and so on. And I could ask a question, how the database management system which is a name of the program that is using the database file, I could ask a question such as how many people are between the ages thirty-two and thirty-five. And in order to answer this question, the database management program may have to look at arbitrary elements or to in order to answer arbitrary questions which could be asked by users, the database management program may have to access arbitrary bytes of the program in some unpredictable fashion as far as the program itself is concerned, and highly dependent on the nature of the queries, the nature of the question that has been asked.

So, in general if one had to characterize the way in which a database file is accessed, one could not talk about sequential access, there will be certain queries for which the access may end up being sequential, but for arbitrary queries, if we would not be possible to actually say that the access pattern in terms of the way in which the bytes of the database file are accessed, could be sequential and one have to say that in general, for database files, one may have to assume that the accesses are random. So, in the best case, one could have sequential access, but other than that one may just have to assume that the access to the different bytes of the file are in some random. So, these are the two terms which are used for commonly occurring file access patterns.

(Refer Slide Time: 19:00)

Now, we want to understand more about what is happening in the operating system as far as the management of a file system is concerned. When I use the word file system, I am referring to a collection of many files. The part of the operating system which manages files is refer to as a file system and that part of the operating system is going to manage not only the files of me as a user, but also the files of all the other users on the system. So, all of those files together come under the preview of the file system.

Now, when we talked about virtual memory, we talked about the design issues from the perspective of the operating system and by doing that, we got a better idea about what happens when our program runs. If you understand how operating system makes decisions about sharing the resources of a computer system, then we can better tune our program to benefiting from those decisions.

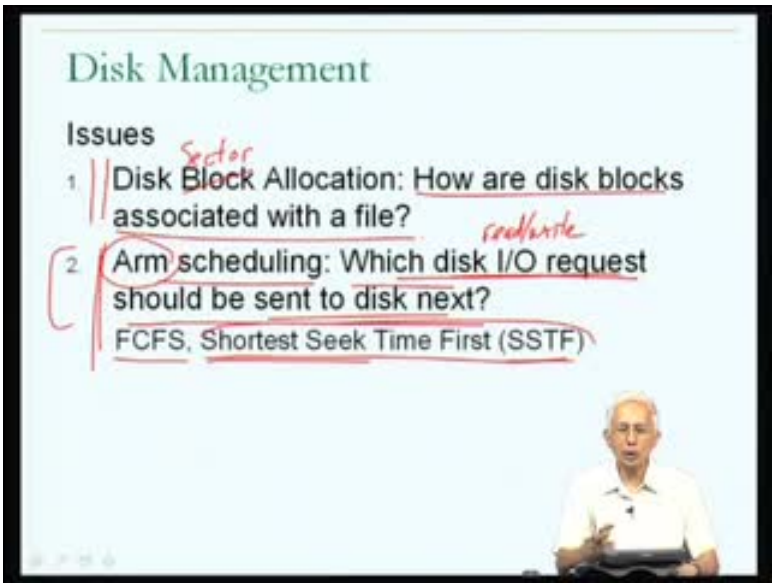
So, will do the same with a file system and try to understand how the operating system manages the files. So, there are basically three main issues as far as file system design is concerned. One relates to how the operating system manages the disk. We see what the disk is like and we understand that the files are going to be stored on the disk and therefore, it will be useful for us to understand how the operating system efficiently uses the disk space for storing all the files that need to be kept track of and that comes under the topic of Disk management.

Another aspect of file system design is what is known as Name management. We know that each file must have a name and that the user is going to refer to the file by the name, but the operating system is storing the information which is contained in a file on the disk where there may not actually be a name stored on the disk. And therefore, the Name management refers to how the names are managed by the operating system. So, that the user can use a name for a file, but the operating system could actually store the files without having to keep track of the names on the disk. So, that is the second aspect of file system design what it is known as Name management.

The third aspect, an important aspect of file system design is the aspect of Protection and essentially here, by talking about protecting files from users who should not be able to access those files. This is an issue which came up when we talked about virtual memory. I talked about how it was important to protect the variables of one process from another process which is why we had address translation. To start with, each process has it had its own virtual address space and therefore, one process could not easily access a variable of another process.

Similarly, there is going to be a need to make sure that the files of one user are protective from other users to make sure that they cannot, one user cannot misuse the data or file of another user. So, that is the third important aspect of file system design that we are going to look at. We will look at each of these three, one after the other, to get some idea of what current profile systems might be doing.

(Refer Slide Time: 21:57)



Now, will start off with disk management. Disk management was the first of the three issues that I talked about. And as far as Disk management is concerned, one of the sub-issues that is of relevance relates to disk block allocation. I am using the word block here, remember, the block and disk sector are synonymous. And I think in this set of slides, I tend to use the word block more. Just remember that, this sector and this block are meant to be synonymous.

So, the question in disk sector or disk block allocation is, for a particular file, how does the operating system associate disk blocks with the file. Another aspect of a Disk management that I have, in some sense, are used to earlier, relates to the disk arm scheduling and this basically relates to the question of which disk input output request, what I mean by disk I O request is a read or write request from the disk and that will often be referred to as a disk I O request- input output request.

So, among all the many possible disk input output requests that may have accumulated, which one should be sent to the disk next. Now, I am not going to talk a lot about the second. Therefore, I will make comments about it right now. We should realize from what we discussed so far that the disk is operating on a time scale which is much slower than the time scale of main memory, little on the processor.

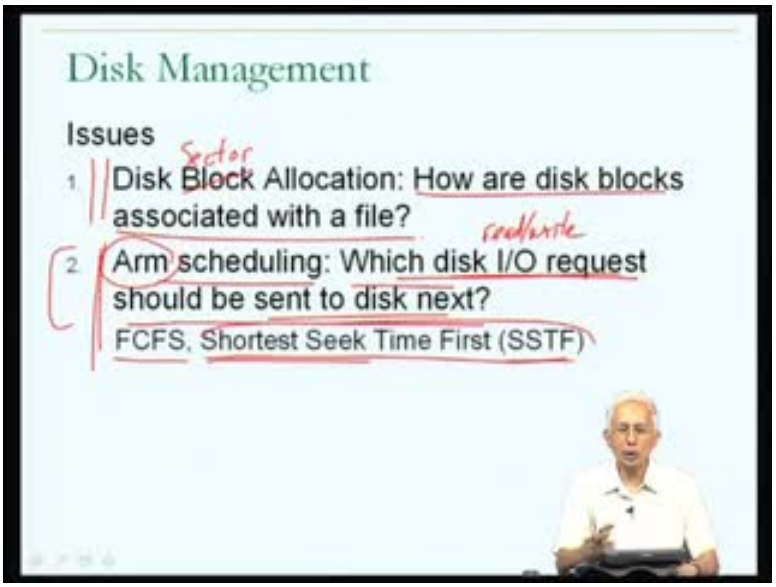
We talked about how the disk access could take ten milliseconds to complete given the kind of seek latencies and rotational latencies that we estimated. What is this mean. This means that if disk accesses are generated because of request from processes and the processes are running on a processor which is running on a nanosecond time scale, then there could be a huge pile up of disk access requests by the many programs in execution, which means that there could be a queue of requests for disk I O operations from the many programs in execution.

Hence, the question arises among the many possible disk I O requests which have piled up for the operating system to send to the disk. Which one should it send to the disk next. And very clearly this must be relating to the movement of the disk arm which is why the word arm comes into the picture. And one can understand that since the seek latency; the seek is the amount of time that it takes to move the disk arm from its current location to the location where the next sector is to be read from.

Therefore, if one's objective is to optimize the amount of movement of the disk arm that takes place, to try to reduce the average amount of movement of the disk arm that takes place, one could do this by re-ordering the requests that have piled up, may be sending the next request which has the shortest seek time first.

So, the operating system could implement various kinds of policies to try to optimize the scheduling of the movements of the disk arm and this would help in improving the average amount of time for a disk access and therefore, the amount of time for file system operations.

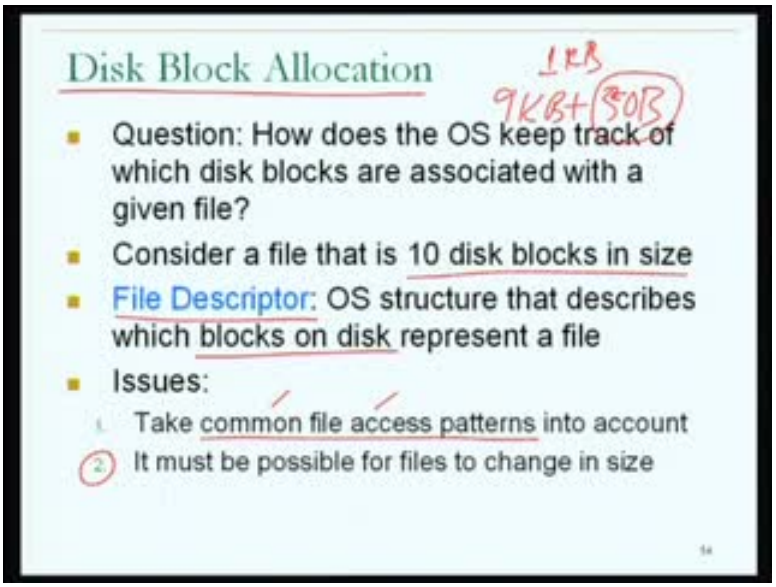
(Refer Slide Time: 24:57)



Therefore, one would hear about various kinds of disk arm scheduling policies such as first come first served, in which the operating system is making no attempt to optimize or reduce the average amount of seek time. Alternatively the operating system might actually keep track of where the disk arm currently is and then try to schedule among all the disk requests which are piled up, schedule the one for which the amount of seek required is the least. In other words, the disk arm has to be moved the smallest amount in order to satisfy that request. So, SSTF are Shortest Seek Time First is another possible policy. And there are more sophisticated policies where for example, the operating system might try to avoid moving the direction of movement of the disk arm. Because if the disk arm is moving in a particular direction and then the next request is in the same direction, it may take less time to move even a few more tracks in the same direction rather than changing the direction of movement when it was moving forward, to make it move backward.

Therefore, there could be more complicated disk arms scheduling policies that could be implemented, but ultimately the objective might be to try to reduce the seek time, ultimately to reduce the amount of time that it takes to read a sector of the disk. So, this is an aspect of Disk management that operating systems may pay attention to. But we will concentrate on the disk block allocation aspect of Disk management. And let me just remind you that the issue there is, if for a given file, how are the disk blocks associated with the file, how are the blocks associated with the file actually located on the disk.

(Refer Slide Time: 26:42)



Now and this is the question. How does the operating system keep track of which disk blocks are associated with a given file. So, if you consider, let us consider a small file. So, I can talk about the size of a file in terms of the number of disk blocks that it occupies. For example, if I told you that the size of the disk block is 1 kilobyte and have a file of size, let say, 9 kilobytes and let suppose that the file of size 9 kilobytes, I then add a little bit of information to it so that it is 9 kilobytes plus about 50 bytes. So, this is something which is bigger than 9 kilobytes, but smaller than 10 kilobytes, then it would actually require 10 disk blocks to store this file; the first nine disk blocks might be full and the 10th disk block may not be full, but this is a file of size 10 kilo of size 10 disk blocks the actual size of the file is 9 kilobytes plus 50 bytes whatever that is.

But I will describe this as a file of size ten disk blocks. Now the question is, how does the operating system keep track of which ten disk blocks are associated with this file. Bear in mind that, it must keep track of which is the first disk block of the file, which is the second disk block of the file and so on, because we are viewing a file as having byte offsets there may be a request to access the thirteenth byte of the file and so on. Therefore, the exact location of each byte of the file must be kept track off.

Now, typically the operating system will keep track of the information about a single file in one data structure. And I will, for the moment refer to the data structure that is used by the operating system to keep track of the information about one particular file by the

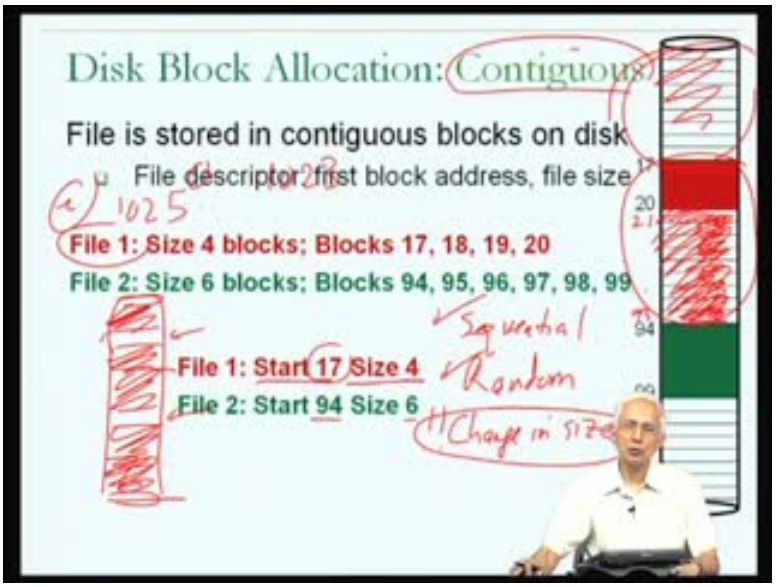
term file descriptor. So, file descriptor I am going to use to refer to, the name of the data structure that the operating system uses and this data structure will contain information about which blocks on the disk contain the data associated with that file.

Now, the issues in doing disk block allocation; and we are currently talking about disk block allocation. What are the issues in the operating systems decision about disk block allocation. Now, the first thing that the operating system must do is that it must use a disk block allocation policy for which the common file access patterns can happen quickly.

What are the common file access patterns? We have talked about sequential access and we have talked about random access. Therefore, in considering many disk block allocation possibilities, the operating system must take into account, the operating, the person designing the operating system must take into account, trying to make it fast for sequential access and trying to make it not so slow for random access as well.

So, that is one issue that must be taken into account in evaluating a disk block allocation idea. The other is that we must bear in mind that files may grow and shrink in size and therefore, if one should not assume in making a disk block allocation policy that files are a fixed size that a file, once it comes into existence will remain the same size forever. Therefore, this is a another important piece of information issue that must be taken into account in designing and file disk block allocation policy within a file system. Bearing in mind that files make change in size; not only become bigger, but potentially become smaller. With this we look at a few possible disk block allocation policies starting with one which is known as the contiguous allocation policy.

(Refer Slide Time: 30:11)



Now, as a name suggest, the idea in contiguous disk block allocation is that a file is stored in contiguous blocks on the disk. And you will remember that our picture of the disk now is of a linear address space. There is disk block 0, disk block 1, disk block 2 and so on. And therefore, when I talk about contiguous blocks on the disk, I am talking about neighboring blocks in this diagram, in the linear diagram that we have on the upper right hand part of the screen.

So, let us just look at an example. Let us suppose that I have file, I will refer to it as file number 1 and this file is of size 4 blocks. So, the file contains many bytes of information, but it requires 4 blocks to contain all its information.

Now, under contiguous allocation, this particular file might be stored in disk blocks 17, 18, 19 and 20 such as shown in the diagram over here. Because 17, 18, 19 and 20 are contiguous blocks on the disk. I could have another file, let say file 2 which is of size 6 blocks and it might be stored in the contiguous blocks 94 through 99. This is the idea of contiguous allocation. And this seems like a good idea, but we do need to evaluate this idea and therefore, we will try to evaluate it. Well, before talking about the evaluation, let me just also sum up, what information will have to be stored by the operating system in the file descriptor for each of these files. In other words, what is the information that must be kept track of in the file descriptor to keep track of where file 1 is located. And

under contiguous allocation, you will understand that, all that the operating system has to keep track of, the starting block of the file as well as the size of the file.

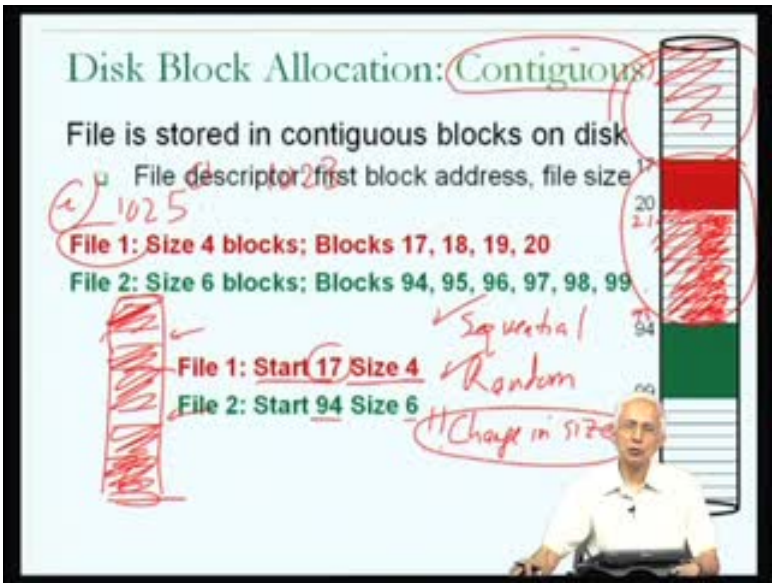
It does not have to remember that the file occupies block 17, 18, 19 and 20. It just has to remember that the starting block is 17 and that the size is 4.

Similarly, for a file number 2, it just has to remember that the starting block is 94 and that the size is 6. Therefore, the size of the file descriptor is going to be fairly small. It is going to have to have one field which contains the starting block and there are another field which is the number of blocks constituting that file. Therefore, whether I have a very small file or I have a huge file, the size of the file descriptor is going to be the same.

Now, how do we evaluate whether this is a good idea or not. Remember that we had two criteria- one was we wanted to make sure that this particular allocation policy is good for the different kinds of commonly occurring file access patterns and we know, of the two, there is sequential and there is random. So, we have to ask how well does this kind of allocation work for a sequential access files and for random access files. The other criteria that we had to take into account was we want to make sure that it is possible for files to change in size. So, these are the three criteria that we have to discuss in talking about in trying to evaluate this idea.

Let us first think about sequential access. So, under sequential access, the idea is, if once a file is opened, the program which is doing sequential access of the file will be the first byte, then the second byte, then the third byte and it will keep on going until it comes to the last byte of the file. And if you think about this a little bit is fairly clear that, under contiguous allocation since the blocks of the file are neighboring to each other on the disk, then sequential access is always going to be easy to do. In order to read the first byte of the file, one merely reach the first block of the file the operating system and then it reads the bytes out of that first block after that, it reads the second block of the file and so on. Therefore, there is no problem with sequential access as far as contiguous allocation is concerned.

(Refer Slide Time: 30:15)



What about random access? In other words, what if the first request that comes to operate on this file is to read byte number 1025 from the file. How does the operating system find out where byte number 1025 is located? Now, we know that the bytes of the file are number from 0 up to the last byte in the file and therefore, if the size of one block is 1 kilobyte; that means, that the first block of the file will contain bytes number 0 up to byte number 1023. And therefore, if I was told that I want to access byte number 1025 or some arbitrary byte number, it is fairly clear that the operating system can calculate which block contains that particular byte.

Therefore, regardless of what which byte the program might be requesting, the operating system can do a very simple calculation based on the starting address and the size of each block to find out exactly which disk block contains that particular byte. And then it can read that corresponding disk block. And therefore, there is no problem as far as random access is concerned. The operating system can, with a very simple calculation, find out exactly which disk sector has to be read in order to satisfy that read/write request.

So, contiguous allocation is very good for sequential access; no problem as far as random access is either, if you think is looking good so far. Third requirement was that it should be possible to change the size of a file. For example, what a file 1 is being used by a program and the program writes more data into the file. Can file 1, can this particular

allocation strategy accommodate a growth in the size of the file? The answer it looks like there is no problem, if this file number 1 has to grow, then it can grow to include block number 21.

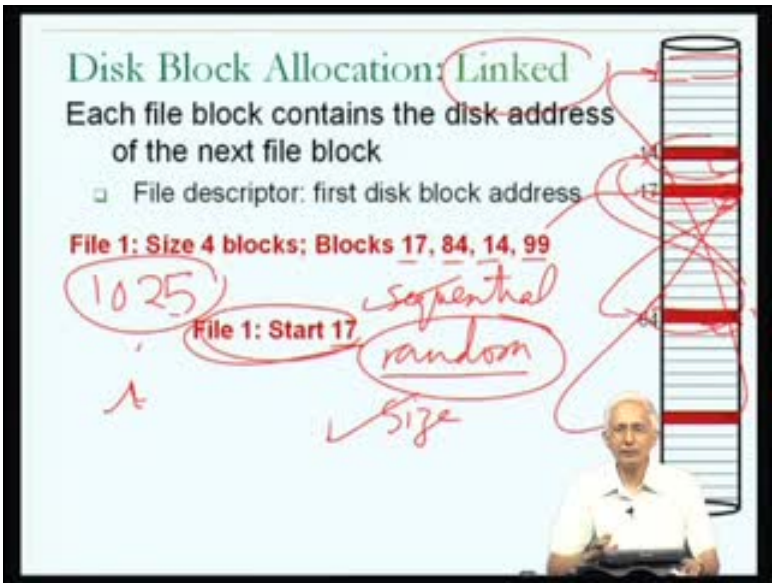
Now, does this argument satisfy entirely? The answer is that, well it looks from this example that, we have a disk which is hardly used at all. But even if you looked at the example of file number 1 growing until late hit block number 93.

So, there were lots of writes to file number 1 and kept on growing until late grow up to a block number 93 and that this point, it became necessary to write a few more bytes into the file. And at this point we notice that it is not possible for file number 1 to grow any more under contiguous allocation because, block number 94 on the disk is not available for use by that particular file.

And therefore, there is a problem with changing the size of the file. What will have to happen for the operating system to allow file number 1 to grow beyond that size. The operating system will actually have to copy file number 1 from its current location to some other location and then allow it grow from there and that would involve a lot of disk activity. Note that, each of the disk blocks would have to be copy from its current location to its new location. And therefore, to accommodate changing sizes of files may actually become a problem.

So, contiguous allocation is not a very favorable policy from the perspective of our second requirement; requirement of files being able to change in size. You will notice that in general, if I have a file system which is fairly full, then there could be a lot of blocks in the file, a lot of occupied disks, a lot of occupied sectors in the disk. If I was to draw the disk again, I may find out that there are a lot of files on the disk and the bulk of the disk may be occupied leaving only a few small gaps of unused sectors on the disk. And suddenly if I want to create a new file, it may not be possible to just create of file of sufficient size using the sum of all these small areas. Therefore in general, continuous allocation suffers in a very serious problem as far as our second requirement is concerned, and it is not typically used by file systems for this reason.

(Refer Slide Time: 37:47)



Now, an alternative to contiguous allocation, something known as linked allocation and the idea in linked allocation, as a name suggest is that, within each file block on the disk,we have a link to the next file block. In other words, the address of the next disk block of that file is stored in the current disk block of that file.

So, let us just look at an example to understand what this means. From the name we understand that, essentially there as a link list connecting the different disk blocks associated with a file.

So, consider for example, file number 1 which once again is of size 4 blocks, but rather than being contiguously allocated, I have allocated it to occupy block number 17, followed by a block number 84, followed by block number 14, followed by block number 99. So, the first block in the file is block number 17 and if you look into block number 17, you find out that there is a link to block number 84. So, at the end of block number 17, there is information that the next file block is block number 84. So, in some sense, this is a link to block number 84. And if I look at the end of block number 84, I find the number 14; it is telling me that the file continuous in block number 14.

And if I look in block number 14, I find that there is link to block number 99 which tells me this file continues in block number 99 in a sense. Each of the file blocks contains a link to next block in the file and hence there is link list of file blocks. So, we have file 1

which occupies 4 blocks, there could be arbitrary file blocks on the disk block number 17, followed by disk block number 84 and so on. What information should I have to keep track of in the file descriptor? Well, to keep track of information in the file descriptor just like for a link list, all that I have really have to keep track of is the first block in the file. So, block number 17 is the first block in the file. Technically that is the only information that would have to be kept track of. Therefore, the file descriptor could be very small whether the file is a huge file of thousands and thousands of blocks or the file is a small file of only 1 block, the file descriptor will be have a very small size.

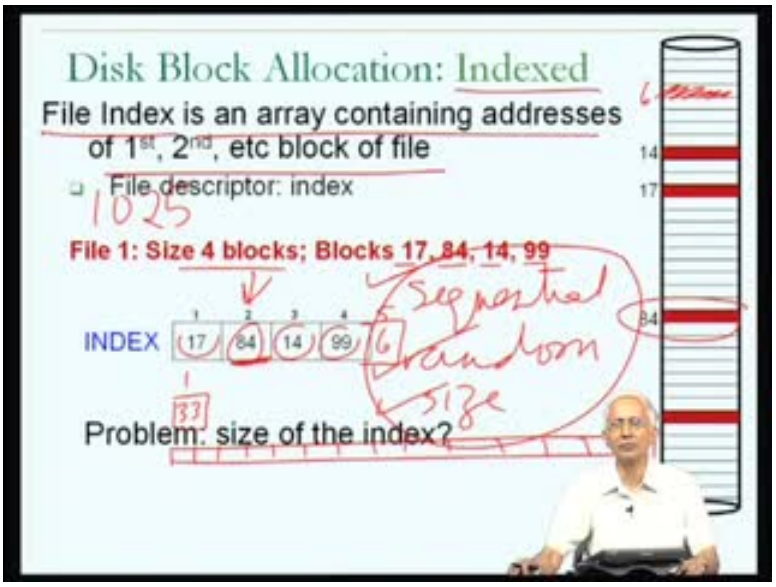
Now, we need to evaluate this allocation policy; the linked block allocation policy from the perspective of sequential access, random access and change in file size. And if you think about the last requirement change in file size, we see that link allocation is a winner over contiguous allocation. You can imagine that, for any file to grow by one more block, all that I have to have is one free block somewhere on the disk and then I could link the last block in the file to this new block on the disk. And therefore, changing the size of a file is extremely easy under linked allocation, unlike under contiguous allocation. In fact, the reason that that the idea of linked allocation came up was as a direct result of the failure of contiguous allocation to allow for files to increase in size, if efficiently.

So, linked allocation is a clear winner as far as change in size is concerned. But what about sequential access, in other words, if I want to read a files starting with this first block and then reading all the bytes in the first block and then moving to the second block, how well would linked access go. And the answer is, it works quite well because starting from the file descriptor, you go to the first block and then you read all the bytes in the first block, and by the time you come to the last byte in the first block, you know which is the second block of the file you therefore, read the second block in the file by you, I mean the operating system and then read sequential the bytes of the second block etcetera.

Therefore there is no problem with sequential access as far as linked allocation is concerned. It can be as efficient as the contiguous allocation scheme. The operating system will be able to very efficiently allow a file which is being accessed sequentially to be access efficiently. What about random access. What if there is a need to access 1025th byte of file number 1. Here suddenly we hit a problem. In order to access the 1025 byte

of file 1, we actually have to start from the file descriptor and read the first block of file 1 of that particular file, find out where the next block is located and then in fact, follow the link list, until we come to the block which contains the required byte. In effect, unlike contiguous allocation where one could do randomly access using a very simple calculation here in order to do to random access to some arbitrary ith byte of the file, you would have to start reading from the first block of the file, going to the second block of the file, then going to the third block of the file etcetera, until we come to the block of the file that contains the required piece of data ,the ith block. And this would could require multiple accesses of the disk each of which is going to take about 10 milliseconds. And therefore, we must say that linked allocation is a complete failure as far as a random access is concerned.

(Refer Slide Time: 43:00)



So, we have seen contiguous allocation which was a failure as far as file size growth is concerned. We have seen linked allocation which is a complete failure as far as random access is concerned. And we need to have an alternative which has neither of these problems and that is the third allocations scheme. This is something known as indexed allocation.

Now, the idea of index allocation is that, it will use something very much like linked allocation in order to avoid the problems with growth in the size of the file, but it will use something very much like contiguous allocation in order to avoid the problems with the

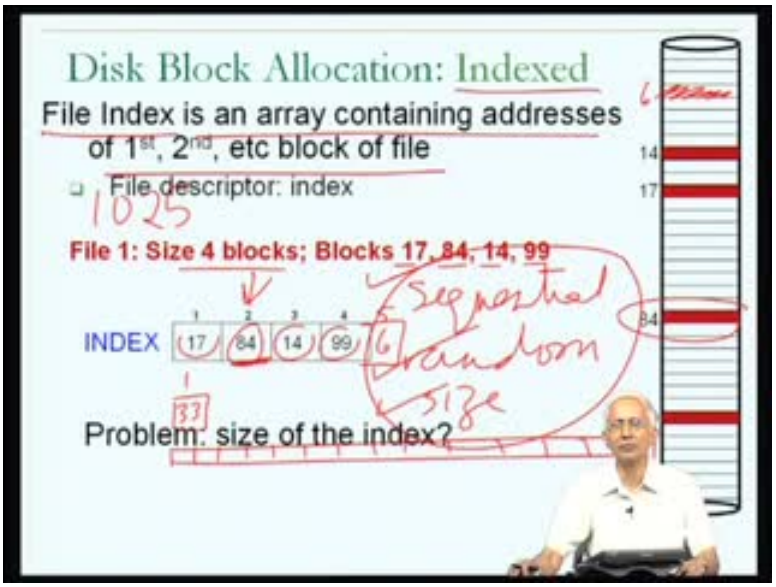
random access to elements of a file. And the way that it does this is by maintaining something called an index as a name suggest or a file index. And what is the index of a file? The index of a file is basically an array that contains the addresses of the first, second, third, fourth etcetera blocks of the file in order.

So, I will show an example of an index .Let us consider for example, file number 1 which is the same file number 1 from a previous example from a linked allocation. So, file number 1 is a file of size 4 blocks and to get the benefits of file growth, I want to have this kind of linking. So, I am going to assume that the first block of the file is block 17, the second block is block 84, the third block is block 14 and the final block is block 99, just as we had in the case of linked allocation. But to avoid the problems with having to read each of the blocks in order to get to a particular file, we will maintain the links in a central location which is what the index is.

So, the file index as I mentioned, is an array containing the addresses of the blocks of the file. So, the index for file number 1 is going to have these four elements in it. A clear indication that the first block of the file is block number 17, second block of the file is block number 84 etcetera. Now, what is the advantage of this scheme? How well does this scheme work for sequential access? How well does this scheme work for random access? How well does this scheme work for changing the size of a file? Well, let us first think about changing the size of a file. The answer is, this is going to be as good as linked allocation was, because if I wanted to add one more block to the file, I just have to add it to, I can use any block on the disk and then I just have to add it as the next element for example, if I add block number 6 as a next block disk block associated with this file, it just had block number 6 as the 5th block in the index of this particular file.

So, it is going to, as well as linked allocation for the size problem, it is going to do as well as linked or a contiguous allocation for sequential access to the file what about random access to the file? But if I want to access byte number 1025 of this particular file, now once again using a very simple calculation, the operating system can calculate that . Byte number 1025 of this file is going to be present in the second block of the file and then by just looking into the index, it knows that the second block of this file is in disk block number 84 and it can directly read disk block number 84 without having to go through the links along the way, as was the problem with index linked allocation.

(Refer Slide Time: 42:54)

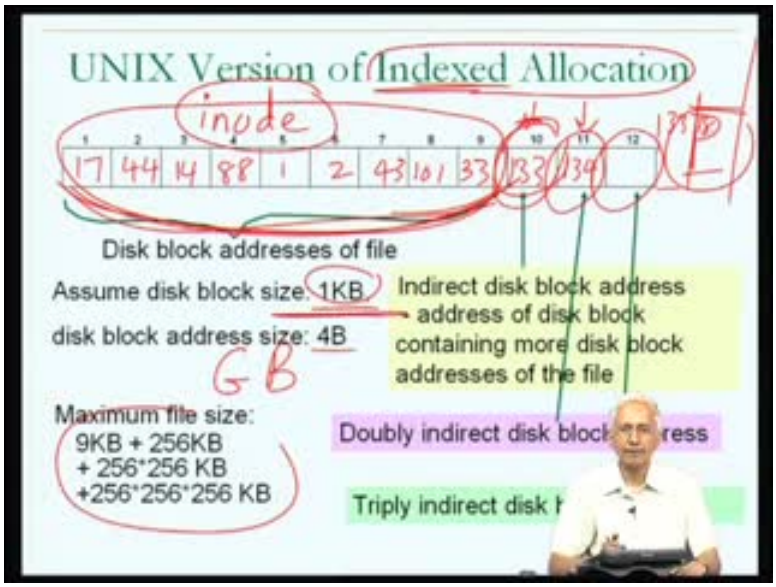


Therefore, index allocation is a winner as far as random access is concerned. Two, it is the first of all strategies which wins on sequential access random access and on the growth of the file. What then, is there any problem with index allocation? The answer is, let us think a little bit about a small file and a large file.

How big will the index of a small file be? Let suppose there is a file which is only 1 block in size, the index of such a file need contain only that 1 block address. What if there is a huge file which is thousands and thousands of blocks in size? Then that particular file must have an index which contains thousands and thousands of disk block address. This then is a problem with index allocation. The size of the index and therefore, the size of the file descriptor. Until now, this was not an issue. When we talked about contiguous allocation, the file descriptor was small. When we talked about linked allocation the file descriptor was small.

But when we add the idea of an index suddenly, the file descriptor itself for a very large file could be enormous and this is a problem which must be addressed by in implementation of indexed block allocation. Because it look like indexed allocation is winner from the other considerations. And one must try to come up with schemes through which this problem with the index size can be overcome. So, the problem with the size of the index is actually overcome by Unix and Linux systems in a somewhat interesting way.

(Refer Slide Time: 47:28)



So, I will just refer to the unix version of indexed allocation through which the problem with a size of the index is overcome. Now, what is used in the case of the version of unix that I am talking about is index, but the index is not going to depend on the size of the file rather the size of the index is going to be fixed. For example, in the version that I am going to talk about, the sizes of the index of any file is 12. In other words, there is space to hold 12 disk block addresses and this is good, the same whether the file size is small or the file size is big. How does this version of Unix manage with such a small index, I mean what is there is a file which is of size 100 blocks, how does it mange with only disk to twelve disk block addresses.

Now, the way that it manages is why an interesting idea. This version of unix does not use all of the disk block addresses the same way. It uses some of the disk block addresses. For example, may be the first 9 to actually store the addresses of the first 9 blocks on disk of the file. So, for example, if the first block of the file is in disk block number 17, then it remember 17 in the first field of the index. If the second is in block number 44 it remembers 44 and so on.

Now, what if the file is bigger than 9 blocks in size? Then, this version of Unix will go and use the 10th block address, not to contain the address of the 10th block of the file, but to contain the address of a block on the disk which contains the next few addresses of file blocks. So, this idea is called an indirect disk block address and the idea, as I said

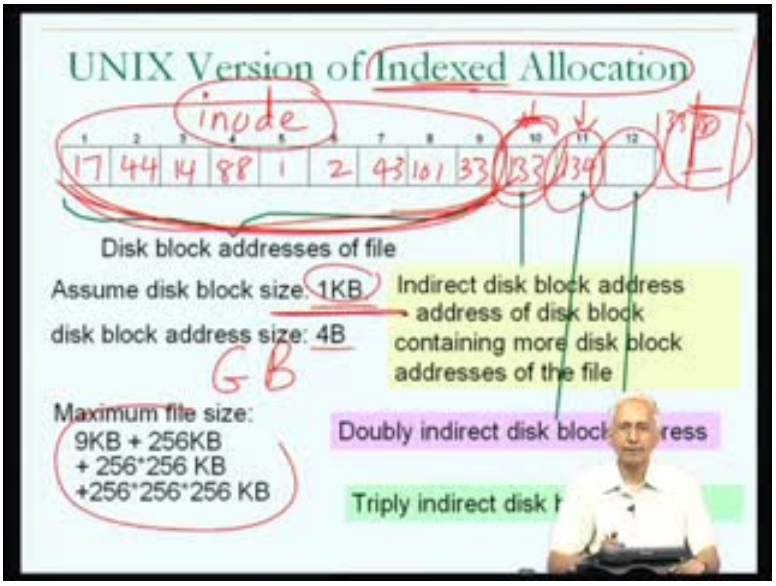
before is, if the size of this file was anything up to 9 blocks, then the blocks could be, the addresses of those blocks could be mentioned in these addresses address locations. If there is a need for one or two more disk blocks because a file was bigger than 9 blocks, then the way that this version of unix would handle it is, it would remember the additional addresses not in the index, but in a disk block.

So, one of the disk blocks would be, let say disk block number 33, I am sorry 33 is already use. I say 133, would be used as a continuation of the index. Therefore, the 10th disk block address would be in a piece of information that is contained inside the disk block number 133. And the identity of the disk block which contains the extension of the index is what is contained inside the tenth element of the index which is why the tenth element of the index is known as the indirect disk block address.

Now, how many additional disk block addresses can be contained within 1 block in in this manner? The answer is only a small number, but we may want to be able to accommodate disks of arbitrarily larger sizes than that. Therefore, what Unix, this version of unix does is, it uses the next available element of the index, index number 11, for what is called a doubly indirect disk block address. And the idea here is, that rather than containing the address of a disk block which contains disk block addresses of the file, the 11th element of the index contains the address of a disk block which contains the addresses of disk blocks ,which in turn contain the addresses of the disk blocks of the file. And this allows much larger files to be represented using the same sized index. And if this is not enough, it has something called, could have something call the triply indirect disk block addresses carrying forward the same idea.

Now, the question is, if I have a situation where there is this kind of an index which has only 12 disk block addresses associated with each index, how big a file could be represented. And we can do a simple calculation, using some information. So, I will assume in doing this calculation that, the size of a disk block is 1 kilobyte, the sector size. I will also assume that the size of a disk block address is four bytes.I need to know the size of a disk block address in doing this calculation, because some of the disk blocks are going to contain disk block addresses of blocks of the file.

(Refer Slide Time: 47:28)



Now, we know that if I have a file which is relatively small, let say up to 9 disk blocks in size, then it can be represented by putting addresses into the first 9 elements of the index. So, files up to size 9 kilobytes can be satisfied using the first 9 elements of the index. If I have a file that is bigger than that, I will have to use a 10th element of the index, the 10th element of the index will contain the address of a disk block which will contain disk block addresses. Then, how many disk block addresses could be contained within one disk block? The answer is, size of a disk block divided by the size of a disk block address 1 kilobyte divided by 4 bytes.

You can do the calculation. You will find out that is equal to 256. Therefore, the indirect disk block address can contain 256 disk block addresses and therefore, each of those could add 1 kilobyte to the size of my file.

So, therefore, by using the 10th index, we can accommodate files which are of size 9 kilobytes plus 256 kilobytes, in other words up to 256 kilobytes in size. What happens if we go and use the 11th index in addition? You can extend the calculation to find out that we can actually go up to 256 multiplied by 256 kilobytes, but if we go up to the 12th in addition you come up with something which is 256 cubed kilobytes. So, what is the size that we were talking about over here, you could do the calculation and satisfy yourself that we are now in a range of few gigabytes of file size. Therefore, using this scheme I could accommodate file, a single file which is of size few gigabytes in size which is

typically adequate, but if as files get bigger and bigger with increasing demands is conceivable that a version of unix could just extend the size of the index by one more address and suddenly the size of the files could become significantly larger, well beyond the gigabytes in size.

So, this is the kind of a working scheme which is used in Unix and Linux file systems in order to do disk block allocation. They have this notion of index allocation with a fixed size index and the information of this index is usually contained in a data structure of the unix file system known as it is index node over inode. So, one of the key terms in unix or linux might be the term inode where inode stands for the index node indicating that is using some form of indexed allocation.

Now, with this, in this lecture, we have seen that the file systems have three main components- one has to do with a management of the disk, the other has to do with a management of the names, the third has to do with protection. As far as the management of the disk is concerned, there were two important problems, we see operating system has to take into account- one is the scheduling of the disk arm which we spoke about briefly, therefore, which the operating system might cause policy such as Short a Seek Time First to be used. The second which we talked about in more detail was what is called the disk block allocation issue. And we saw that operating systems today, might use something called the index which is a fixed size array containing the addresses of disk blocks associated with a file, in order to make the time for sequential access or random access or the size of the index or the possibility of file growth, all to happen in a fairly efficient fashion. In the lectures to come, we will talk about Name management and protection in a little bit more detail. Thank you.