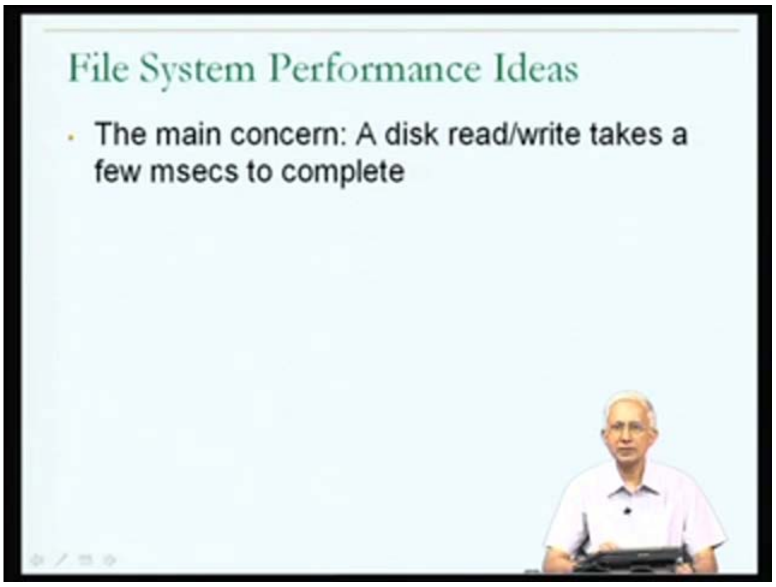
High Performance Computing

Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 08
Lecture No. # 37

Welcome to lecture number 37 of high performance computing. We are currently looking at file systems trying to understand the design principles inside the operating system in providing support for files, files being storage of persistent data typically on secondary storage devices such as disks. Now, we had understood the basic mechanisms of disk block management, name management and protection in the previous few lectures. Today, we are going to concentrate for much of the time on some file system performance ideas, in other words what is going to be important from the perspective of improving performance of programs that deal with files. And the fundamental reason that, this is important needs to be understood first.

(Refer Slide Time: 01:00)



File System Performance Ideas

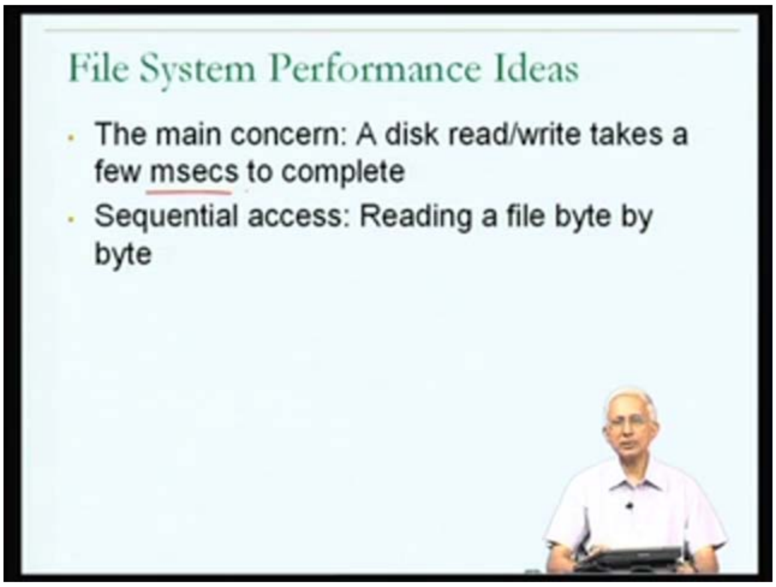
- The main concern: A disk read/write takes a few msecs to complete

So, let me start with the slide where we talk about the file system performance ideas. And why this is the concern? Why is it concern to us? Is it not just enough to know how file systems are designed and use the information correspondingly? Now, the main concern that we have in connection with file systems is the opening comments that I had made about disks. I have talked about the structure of our magnetic hard disk drive and we had done this clear understanding of how it is structured and how much time it takes to access a sector of the disk. The key components of that were the seek latency and rotational latency, but the bottom line was we estimated that a disk read or write takes a few milliseconds to complete.

The number could be a typical number would be something in the nature of 10 milliseconds for a current hard disk, 10 milliseconds for a disk read or write operation to complete. And in writing programs a deal with files therefore, I have to bear in mind that every time that my file does read or write to or from; I am sorry every time my program does a read or write to a file. Given that the file is stored on disk that particular operation within my program is going to take may be 10 milliseconds that read operation or that write operation is potentially going to take 10 milliseconds. And therefore, it is a great concern to us, if I have programs which are doing a lot of file input and output operations.

This could dominate any concern that I may had about main memory. Remember main memory was only 100 nanoseconds compare to the 1 nanosecond speed of the processor when certainly talking about the file I/O operations where to read a byte from the disk could take 10 milliseconds. This is a formal series problem and therefore, very clearly of concern to us. And let us think about this in the connection of one of the common file access patterns. Now, I talked about one kind of typical behavior shown by programs as far as accessing files is concerned.

(Refer Slide Time: 03:00)



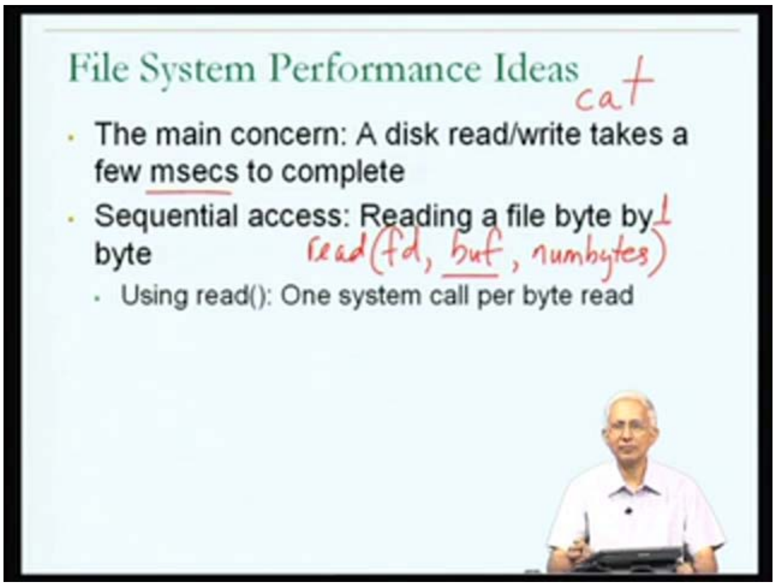
File System Performance Ideas

- The main concern: A disk read/write takes a few msecs to complete
- Sequential access: Reading a file byte by byte

I refer to it as sequential access, the property of a program doing sequential access of a file is that it program opens a file and then reads the file byte by byte starting from the first byte in the file, going to the second byte in the file and bidding all the way through till the end of the file there is what we meant by sequential access. So, if I let suppose that I was writing a program, which had to do sequentially access a file, it is possible that I am writing a program. For example, I might have been given the task of writing a cat program, do you remember I have talked about cat, Catinate? It is one of the programs available on any Unix or Linux system using which you can calls the contents of a file to be printed out.

Obviously, the cat program is opening the file and reading the byte starting from the first byte second byte so on, it reads the byte and then it prints the byte out let say on the screen. So, if I was ask to write such a program, I would very clearly have to worry about the fact that every read that I do, I have first read the first byte and that may taken 10 milliseconds and I printed on the screen. Then I read the second byte and taken other 10 milliseconds and printed on the screen and so on.

(Refer Slide Time: 04:00)



File System Performance Ideas

cat

- The main concern: A disk read/write takes a few msecs to complete
- Sequential access: Reading a file byte by 1 byte
  read(fd, buf, numbytes)
  - Using read(): One system call per byte read

Therefore, this program could potentially take a huge amount of time to print a large file on to the screen. Now, that is just what would happened if I was using for example, the read system call, to read a byte at a time from the file. You will remember that the in the read system call or in the library function that was going to call the read system call for us, you had to mention the file descriptor. In other words that small integer that is writtened by the open system call to specify which file you want to read, then you had to specify a buffer into which you wanted the contents of that the byte information that was read from the file to be stored so, this a variable of your program. And then you had to mention the number of bytes that you wanted to read from the file.

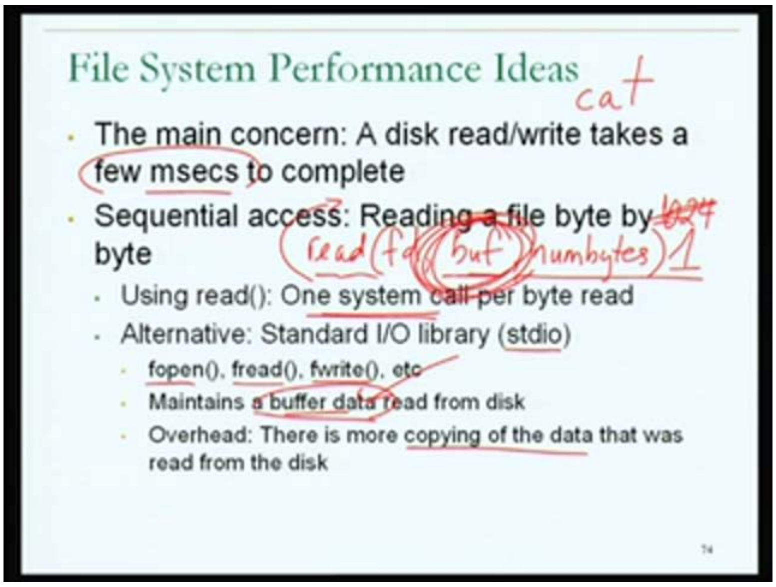
So, if I was doing the cat program, I could set number of bytes to 1 and read one byte from the file and then printed on the screen and then read the next byte from the files. So, I could have the read system call or the read library function call embedded in loop, each time through I read one byte from the file. Now, if this was the case, then I would have one system call executed for every read from the disk; I am sorry for every read from the file. And that one system call would actually end up doing a disk I/O operation and that disk I/O operation could take 10 milliseconds and therefore, this does not seen like a very good way of setting up the program.

So, rather than reading one byte from the file, it may make sense from me to try to read a 1024 bytes from the file, one sector let say 1024 bytes from the file. So, I could improve on the operation of this cat program by increasing from 1 to let say 1024, if I knew that the sector size was 1024. Because by doing this I could read one sector from the file and then printed out in the rest of that loop execution. But even then the cost that I am paying is one system call for every one of those reads. And we had determined when we talking about system calls, that the system call itself is not an operation that comes for free.

Because in inherent in the system call is this situation, where the operating systems switch is mode from running in user mode, to running in system mode and therefore, there is some over head in making a system call. And at one point, I had mention that the overhead could be on the order of microseconds. So, while this is not as bad as the amount of time that it takes to do a disk I/O, it is still much worse than the amount of time that it takes to read something out of main memory. Remember, that was only in our current estimation of things a 100 nanoseconds also, the microsecond is much worse than what we are talking about.

Therefore, using system calls in this way, where I making a system call every time through my cat loop does not seem like a very good idea. What you want to something better than that? Now, those of you who have actually done file programming with files on computer systems, may have known that is an alternative which is provided by the standard I/O library.

(Refer Slide Time: 06:50)



File System Performance Ideas

cat

- The main concern: A disk read/write takes a few msecs to complete
- Sequential access: Reading a file byte by ~~byte~~ 1024
  read(fd, buf, numbytes) 1
  - Using read(): One system call per byte read
  - Alternative: Standard I/O library (stdio)
    - fopen(), fread(), fwrite(), etc
    - Maintains a buffer data read from disk
    - Overhead: There is more copying of the data that was read from the disk

This standard I/O library is available on all the systems that you deal with is typically refer to as s t d I/O, s t d standing for standard, I/O standing I/O library. And the standard I/O library actually provides you with certain functions, which will themselves at some level refer to system calls. But the functions themselves will help a lot towards making the implementation of the cat program better. Essentially what many of the standard I/O library functions do and the standard some of those functions that I am referring to go by name such as, fopen() which can be use to open a file, fread() which can be use to read from a file, f()write which can be use to write to a file and so on so that, the collection of standard I/O library functions of this kind.

And the common property that they have is that, they deal with not individual reads and writes directly from the disk, but they manage a buffer of data which has been read from the disk. So, in association with the program that I have written, if I am using this standard I/O library, then the first time that I use fread(), a lot of information will be read from the disk into a buffer which is manage by the standard I/O library. So, even if I am reading only 1 byte from the file, if I make number bytes equal to 1, then when if I am using fread from this purpose, then more than one byte would be read from the file and maintained by the standard I/O library in a buffer of its own.

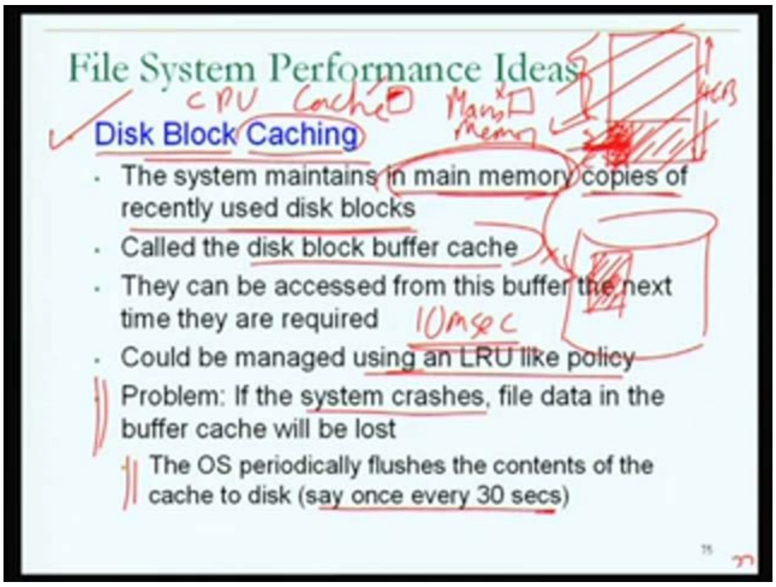
The first byte will be referred to the buffer will be writtened in the buffer that I am using, because that is the way I have set up my program. But the rest of those bytes will be actually will be available in memory in a buffer manage by the standard I/O library. Now, the advantage of this is that subsequent reads that my that my program made can actually be handle out of the buffer the standard I/O library buffer. So, this is the vast improvement of what we had tried before, the number of system calls can be kept down by the standard I/O library prudently it maintaining the buffer well, but there is still a problem in this scheme. In the problem with this scheme is that, they will be more copying of the data, that was read from the disk note that the data was read from the disk into the buffer which is maintained by the standard I/O library.

Subsequently, the data will have to be copied from standard I/O library buffer into the buffer of my program and that would be two memory operations; a memory operation to read from the buffer of the standard I/O library. Another memory operation to write into the buffer of my program and so on. Therefore all of this copying of data is going to involve multiple operations at that 100 nanosecond main memory speed. And is therefore, still has to be viewed as being an overhead, because it involves copying from the standard I/O library buffer to the buffer of my program.

Now, looking back at the system call alternative, the alternative of the standard I/O library is good, but it still has some overheads. And hence when we talk about file system performance ideas, we hopefully learn about mechanisms which are provided within the file system to reduce the times involved in accesses of the kind. That I am referring to here by programs, that I may write even more. Now, I am going to talk about four such ideas and I should just mention that the copying which I am referring to overhead. Refers to the fact that first of all every once in a while the standard I/O library is going to read data from the disk and that is going to involve copying it from the disk to the standard I/O buffer, that is the transfer at the transfer rate of the disk.

Subsequently, every time my program actually does in f read or in fwrite(), the data would be copied from the standard I/O buffer into the buffer of my program, which is multiple main memory operations.

(Refer Slide Time: 10:27)



CPU
Cache
Main Mem
10msec

File System Performance Ideas

Disk Block Caching

- The system maintains in main memory copies of recently used disk blocks
- Called the disk block buffer cache
- They can be accessed from this buffer the next time they are required
- Could be managed using an LRU like policy
- Problem: If the system crashes, file data in the buffer cache will be lost
  - The OS periodically flushes the contents of the cache to disk (say once every 30 secs)

Now, I am going to talk about four key ideas from the perspective of file system design that will be important for a user, who is writing programs that deal with files the first of these is something known as disk block caching. And we have come across this some of these terms before, disk block is our synonym for disk sector, And we come across the word caching from the word hardware cache, the hardware cache which was present between the main memory and the processor. Here, very clearly we are talking about something which is done by the operating system.

And therefore, it must be referring to a cache like idea, maintaining something similar to how a cache hardware cache maintains data, but being done by the operating system. The basically a software idea, that is essentially, what disk block caching is. The idea here is that, the operating system will maintain within main memory so, that is a key observation, copies of the recently used disk blocks. So, when my program first opens a file and then it reads let say one byte from the file that may involve reading the first block of the file from the disk. But if the operating system is doing disk block caching, then it will in effect have a copy of that particular disk block somewhere in main memory.

And therefore, subsequent reads of the bytes of that particular file will happen not out of the disk, but out of main memory, out of this structure in main memory which would call the disk block buffer cache. This is the commonly occurring name in Unix and Linux systems. So, this is our great idea, because it tells us that if for example, I have a program which reads a particular collection of data from a file and then reads the next data from the same file. Then, I can expect that, the accesses to the data need not involve each of the accesses to the data even if they take place through a system call, need not involves separate disk I/O operations. Each of them need not involve a 10 millisecond time of access to the disk.

But rather, it may be possible for the operating system to satisfy that request out of the copy of the disk block which is present in the main memory. You now understand by this called caching, it is very similar to the benefit that one gets by having a hardware cache. Many of the times a piece of data which the processor is trying to access out of main memory, can be accessed out of the cache instead at a much smaller amount of time. So, here we get the same benefit hopefully many of the times the piece of data need not be accessed out of the disk, but can be accessed out of the disk block buffer cache, which is inside main memory. And therefore, accessible in let say 100 nanoseconds rather than in 10 milliseconds, which would have been in the case with the disk.

So now, the data will be available in this disk block buffer cache. In case, it is required again or in case neighboring data from the same disk block is accessed by the program.

Such as, in our example of the sequential access through a file in order to write the cat program, we now see that it does would not have made much of a difference, if I had used one system call in which I read one byte of the file. Because the same information the entire block of the file would now be available, if the system uses disk block caching in the disk block buffer cache for subsequent system calls. And therefore, the benefits of disk block buffer caching are fairly evident to us.

Now, the question which may arise at this point is, where is this disk block buffer cache going to be managed? And we must bear in mind that this is all coming out of main memory, you will recall that we have this general idea that the main memory is on the computer systems that we are dealing with could be something like 4 gigabytes. And until now, we were working under the assumption, that the main memory was used by the operating system to maintain the physical pages. In other words copies of virtual pages of many of the processes which were in execution, which were active.

We now realise that some of the region of the main memory is going to be used for other purposes. For example, we understand that some of the region of main memory might be reserved by the operating system, for use as a disk block buffer cache and the more space that is reserve by the operating system for use as a disk block buffer cache. The more disk blocks can be copied into main memory and available at main memory speeds and therefore, the more benefit they will be to the programs which are doing a lot of file related input and output. And the other hand the trade off that is involved is that the more of main memory that is used for the disk block buffer cache. The less main memory will be available for other purposes, such as store in the virtual pages of the many programs that are in execution.

And hence there will be more situations where those programs will suffer page faults and hence once again the disk accesses will have to be done to satisfy the page faults of those programs. So, there is a trade off that is being played here and the decision about how much space will be made available for the disk block buffer cache, is an important configuration decision that a system administrative will have to make in order to play this trade off. And they would be made based on a typical understanding of the nature of the different kinds of programs and there importance of having a large amount of disk block buffer cache to speed up the programs they do a lot of file input and output operations.

So, it looks like the disk block caching idea is the clear winner. We should bear in mind that since there is a trade off to be played in deciding how big the disk block buffer cache can be. There is a possibility that a replacement policy will have to come into play in managing the current disk block which are maintained within the disk block buffer cache. And off hand one might say that in other situations in hardware and in software where replacement had to be done in L R U like policy made sense because of the principle of locality of reference. But unless, we have a reason to believe that files and disks and the disk blocks show locality of reference. It would be difficult to just assume that an operating system could use an L R U like policy, but in the absence of any model of how file I/O at the localities are the assumption of L R U like policy would not be out of place.

Now, there is one problem, which the idea of disk block caching does raise and we do need to be aware of this. And this problem arises, because until now our picture of the file systems was that a file is stored on disk. And it is stored on disk, because it is important that the data of the file be available in a non volatile and a persistent fashion. And they were two aspects to this, one aspect of it was there could be a program that opens a file and writes into the file. And after the program seizes its execution, the file data still continues to exist. But there was another aspect that I had raised and that was while the program is in execution and reading or writing the contents of a file.

It is possible that, the power is which is keeping the system up and running fails or in general that the system might crash. The system might suddenly terminate correct execution and may be in a power down mode.

Now, if this happens unfortunately, we still want to have files having let me take back the unfortunately even if this happens, we would want the files to have the property that the data that they contain survives. In other words, they should be able to survive the power of the system going down. And that is the reason that we stored the files on something like a disk a non volatile persistent form of storage, which does not require electricity in order for the data to continue to reside correctly on the disk. But if the system crashes and we have a situation where some of blocks of the file are actually available in a buffer cache, then the situation is dangerous.

Because it is quite possible that, the buffer cache contains versions of the file, a blocks of the file which have been modified by the program in execution. And that therefore, the copy of the disk block in the buffer cache could be different from the copy of the disk block on the disk. And the copy of the disk block in the buffer cache might be the more recent one the correct one and the copy of the disk block on the disk might be an older one the wrong one. Hence, when the system crashes and the contents of everything in main memory disappears, the correct copy of that particular file has gone away and what we have on disk is no longer the most recent version of that file.

Hence, it is important to have some kind of a mechanism through which the contents of a file can be cause to be present on the disk despite, the fact that the system has crashed. And one could imagine that; one could try to apply an idea similar to the corresponding situation in the case of the hardware cache. Now, you remember that in the case of the hardware cache, we had the C P U, we had the cache and we had main memory. And we had a problem where they could be a block which was present in main memory which was copied into the cache. And the copy in the cache might get modified by a store instruction. Therefore the copy in the cache is different from the copy in the main memory, the copy in the cache is correct the copy in the main memory is the old copy.

And we talked about how caches could be designed to try to make the cache copy and the main memory copy consistent. And one possibility there was the write through main memory update policy where every time the c p u modifies copy of data inside the cache, immediately the main memory copy is updated. The alternative was the write back policy, but could we use a write through policy in trying to avoid the problem that we have now identified, that arises in file systems, because of disk block buffer caching. In other words whenever the disk whenever the file is modified in its cached copy could be simultaneously modify the copy on the disk.

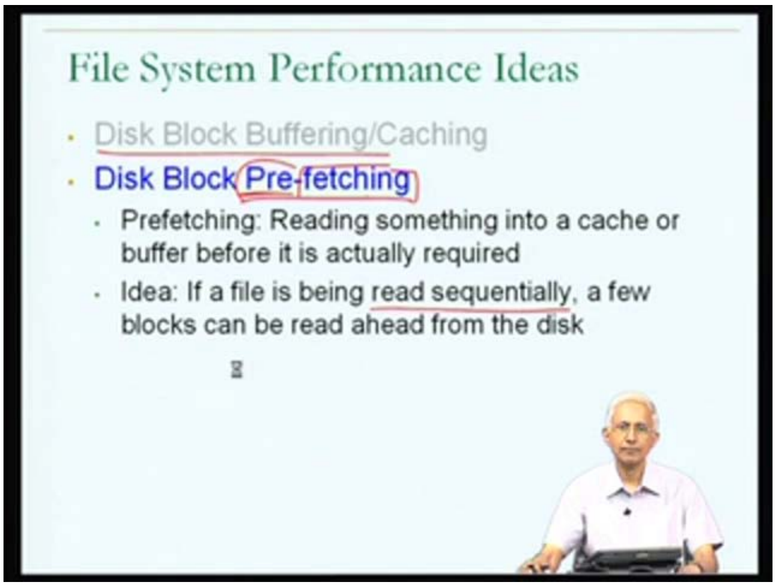
Now, this may not be that good an idea given that every disk access takes potentially 10 milliseconds unlike the situation in the in the main memory cache relationship where the difference in speed was only 100 nanoseconds to a few nanoseconds. Therefore, using write through may not be a feasible option in this context, but the question then arises, what about just using write back? In other words just before a particular disk block has to be replaced updated in the write its current version on to the disk. Now, unfortunately we are not too sure about how frequently elements will have to be replaced out to the disk cache. If the disk cache is fairly large is conceivable that blocks may not have to be written back and replacements may not be necessary very often.

And therefore, it may be necessary to have mechanism which tries to keep the consistency between the disk block, a disk buffer cache copy and the disk copy more frequently. In something which many of the operating systems do is that, they will periodically flush or write back the contents of the cache to disk. And by periodically they will be a parameter of the operating system, the default value may be something like 30 seconds. So, about twice every minute, the contents of the disk block buffer cache would be written into the cache I am sorry would be written back into the disk.

And this is done largely to ensure that at least for files which are not modified more than once every 30 seconds the copy in the disk block buffer cache and the copy on the disk will be exactly the same. And therefore, certain amount of non volatility and persistence of the contents of the file will be ensured. So, this is an important setting; important property of the operating system, for any operating system which is implementing disk block caching. Now, you can assume that all the operating systems that you are dealing with Unix, Linux, windows and so on do use disk block caching. Because the alternative of not using disk block caching would mean that every access to a file would take that 10 milliseconds that we were worried about.

Therefore, this it is quite fair to assume that the system, that you deal with do have disk block buffer caches and therefore, it is an important idea.

(Refer Slide Time: 22:13)



File System Performance Ideas

- Disk Block Buffering/Caching
- Disk Block Pre-fetching
  - Prefetching: Reading something into a cache or buffer before it is actually required
  - Idea: If a file is being read sequentially, a few blocks can be read ahead from the disk

Now, with this one idea let us move on to another interesting idea. Now, the second idea is something which is known as disk block pre-fetching. And that is an interesting term, because if come across the word fetching before we had this notion of part of the instruction execution pipeline we refer to as instruction fetch. And it was the instruction fetch hardware that fetched an instruction from main memory into the instruction register. Here we have; so, we understand that fetching must be getting something from one place and copying it into another place. In this case we are talking about disk block so, we suspect that disk block pre-fetching must refer to copying it from the disk possibly into a disk block buffer cache, but here the word pre or the prefix p has appeared.

And the immediately what comes to mind is, the concept of before there is a need to fetch a particular disk block from the disk into the main memory. In other words before there is a need to fetch it actually bring it into the main memory. In other words this is some kind of a preemptive or speculative kind of an activity where even though the program has not ask for a particular disk block, the operating system if it is doing disk block pre-fetching, the operating system may initiate the fetching of a disk block into the main memory even before the program requires it. So, that seems to be the implication of the of the term disk block pre-fetching.

And that is in fact what I am going to define it as, pre-fetching is the operation of reading something into a cache or buffer before it is actually required in this case before it is actually requested by the program which is doing the file input and output operations. Now, the question which will arise is under what conditions could the operating system profitably pre-fetch a disk block from the disk into the disk block buffer cache? And the answer is if you think about sequential access. In other words, if there is remember the sequential access was one of the two common modes, the common access patterns that we talked about.

The idea of sequential access was that, there is a program which is reading a file, it starts for reading the first byte, then the second byte, then the first byte it goes to the file, sequentially from beginning to end. Now, if there is a program which is sequentially accessing a file, then it should be quite clear that, by the time the first block of the file has been read the is fairly sure that the second block of the file will be required. And therefore, the operating system could have initiated a pre-fetch for the second block in the file, while the accesses to the first block in the file would happening out of the disk block buffer cache.

And in fact the operating system, if it has information, if it is able to keep track of the nature of operations on the disk remember, that the operating system is in some cases involve with scheduling the disk arm movements. It knows what are the different request which have piled up for the disk. So, if it notice that there is not very much activity as far as the disk is concerned, in other words the programs which are currently in execution are not doing a lot of I/O operations required the disk. Then the operating system could use that as an opportunity to pre-fetch additional blocks from the disk into the disk block buffer cache. Anticipating the sequential read requirements of the programs in execution based on the disk block which they have currently accessed.

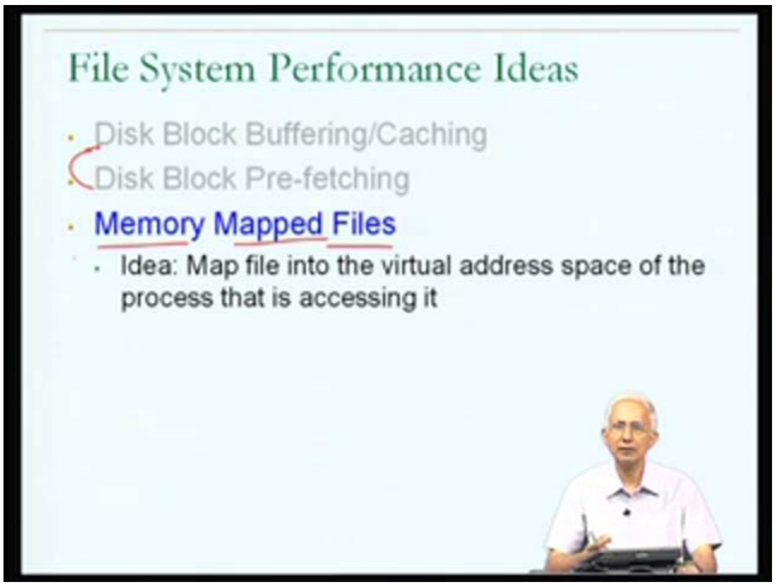
Assume that, they are in the near future going to access the neighboring disk blocks out of the disk so, that is the premise of the idea. And any operating system which is able to do this will be able to provide substantial benefits to the programs which are doing sequential access. The net effect is going to be that the program may never see a 10 millisecond gap during which a disk I/O operation is taking place. Because the operating system may have consistently been able to pre-fetch the blocks before the program actually indicates a need for accessing those blocks. And therefore, sequential access could conceivably take place without ever seeing the disk I/O over heads at all.

So, disk block pre-fetching seems to be a good idea and they will be certain operating systems in which it will be possible for the programmer to specify in the forms of hints to the file system. Information about whether or not the program is going to be making sequential access, these would be known as hints. Since, the operating system cannot be sure that the programmer is going to have detail knowledge about how the program is behaving, but sophisticated programmers may be able to give the appropriate hints. And then, the operating system could use those hints by doing pre-fetching.

You will notice that pre-fetching of a disk block into the buffer cache it is not a going to effect the correctness of a program, the program would have run correctly. In other words would you have produce the same results whether or not the pre-fetching was done. But the pre-fetching may have a substantial impact on the performance of the program program may be able to run substantially faster if the operating system was able to successfully pre-fetch the blocks that the program is going to access from its various files. So, disk block pre-fetching seems like a simple idea something that an operating system could quite easily do and something which would be a read benefit to the programs. Further it is conceivable to imagine that the operating system could be keeping track of the order in which the bytes of a file are being accessed.

And therefore, anticipate whether it would be beneficial to pre-fetch subsequent blocks of that particular file.

(Refer Slide Time: 27:30)



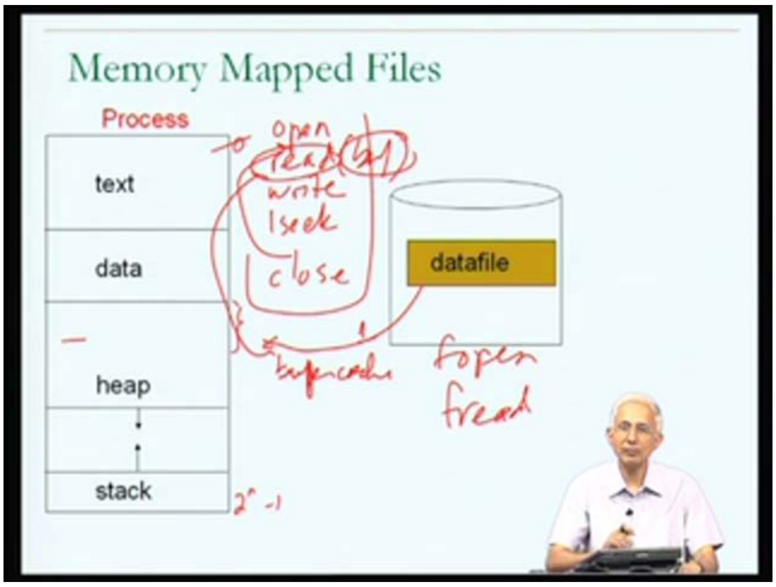
File System Performance Ideas

- Disk Block Buffering/Caching
- Disk Block Pre-fetching
- **Memory Mapped Files**
  - Idea: Map file into the virtual address space of the process that is accessing it

So, disk block pre-fetching seems like a good idea and somewhat easy to implement. It is allied to the idea of disk block buffer caching which we suspect is going to be present in any operating system, certain aggressive operating systems are also likely to do disk block pre-fetching. Now, moving on to another idea, I am going to talk about an idea which is known as the idea of memory mapped files and again just looking at this term from a distance without technically. We understand that this has something to do with the file operations, but it seems to be a situation where there is an attempt to map the contents of files into main memory. In other words to somehow have the files present in main memory rather than having to read the files of disk. But that is that was the idea as far as disk block buffer caching was concerned. Therefore, this must be as some other twist on the use of main memory for holding the contents of files.

Now, the general idea behind memory mapped files is the following. We know that associated with any processes is a virtual address space. Now, if a process is actually handling data which is available within a file and the programmer is aware of this. Then, the programmer should be able to indicate this by mapping the contents of the file into the virtual address space of that process explicitly. If such a facility is provided by the operating system, what does it mean to map the contents of the file into the virtual address space of the process that is accessing it?

(Refer Slide Time: 29:00)



Now, let us just look at this pictorially over here I have a drawing of a disk I am showing the disk in this form, because remember we think of a disk as being made up of many platters each of which has cylinders. Hence, this cylindrical kind of representation for a disk, it is a fairly standard kind of a representation showing a cylinder for a disk. So, let us suppose that I have a disk, which contains one particular data file that a program that I am interested in is going to accesses. And I am not sure if the program is going to access this file sequentially or whether it is going to access it randomly. And in fact the idea of memory mapped files will be able to satisfy both of these requirements very well.

Now, I know that when the program executes as a process, this going to have a memory image, which is going to include its text, its data, its stack and its heap. And we know exactly what each of these components of the memory image of the process is. Now, the idea of the memory mapped file as describing the previous slide was that the contents of the data file would be mapped into the virtual address space of the process. Remember, if the virtual address space of the process goes from some minimum address to some maximum address. And talking about mapping the contents of the file into the virtual address space essentially means that, for all attains a purposes, the range of addresses into which I have done the mapping.

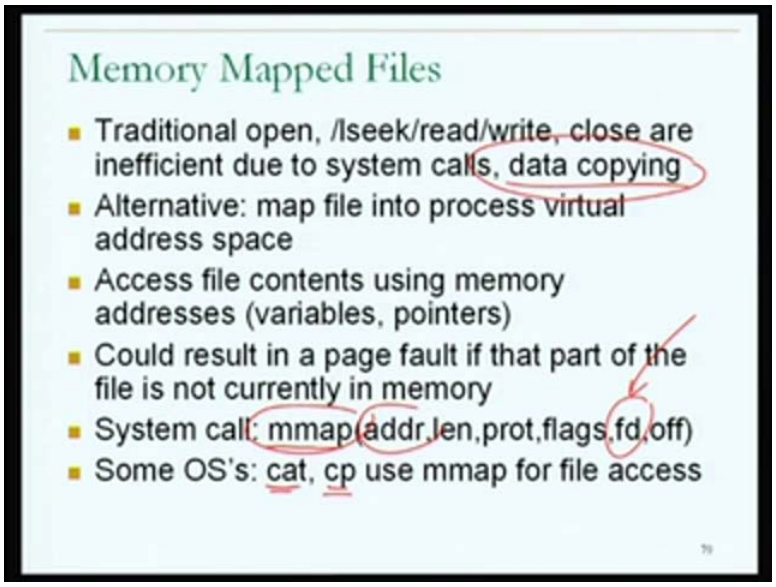
If the program reads from any of those addresses will end up reading the contents of the file that is, but this is the picture that one should have in mind when one things about a memory mapped file. So, that is the file which exists on disk, but the programmer has somehow managed using facilities provided by the operating system to cause the contents of that file to be associated with some region of the virtual address space of the process corresponding which is the program in execution. Now, what is going to be the benefit of doing this? Now, if this had not been done, in other words if one had not done this memory mapping, then the situation would have been like this.

And when the program had to access the data file, if I looked at the program, I would see it would have had to open the data file and then it would had to read from the data file or write from the data file or lseek from the data file. And after having done that whole thing, it would have close the data file after doing multiple read writes and lseeks. And each of these was in the worse case, a system call which would take a few microseconds in the better case it could have been library calls. But in both cases they would have been the need to copy data from one place to another. For example let suppose that I had implemented my program using system calls then every time there was a read, data would be read from the disk into a buffer cache.

So, that is one read into the buffer cache which is within main memory, but outside the virtual address space of the process so, that is one copy of data. Subsequently, they would be a need to copy data from the buffer cache into the buffer associated with the read call. So, two copying two need; two situations where data has to be copied, one from the disk into the buffer cache and one from the buffer cache into the buffer. One is of course, at the disk latencies, but the other is at main memory latencies, but there is data copying that has to be done.

Now, even if I had setup the program to operate using the standard I/O library fopen fread etcetera. Data copying would have been involved not from the disk to the buffer cache and from the buffer cache to the buffer, but from the standard I/O library buffer into the buffer of my program. So, once again data copying would have been involved and that would have been in the case if I had not used memory mapping.

(Refer Slide Time: 29:00)



Memory Mapped Files

- Traditional open, /lseek/read/write, close are inefficient due to system calls, data copying
- Alternative: map file into process virtual address space
- Access file contents using memory addresses (variables, pointers)
- Could result in a page fault if that part of the file is not currently in memory
- System call: mmap(addr,len,prot,flags,fd,off)
- Some OS's: cat, cp use mmap for file access

So, the problem with not using memory mapping is, that I would if have to use a traditional open, lseek read write or f open f read f write fseek etcetera. But all of them are inefficient either because they use system calls which is the case with open lseek read and write or because they involve data copying which is the case with fread fopen etcetera. What is the advantage of using this memory mapped file? If I use the memory mapped file, then I am actually have a situation where the data of the file is available in the virtual address space of the process. As shown by the diagram, the data of the file is available using certain addresses of the virtual address space of the process.

And therefore, the program can access the data contained in the file using memory addresses, in other words using variables or pointers. And you could easily see that in setting this whole thing up the program may have a single pointer which points at the first element of the file and that if it wanted to read the thousandth element of the file it could just move the pointer ahead to the correct byte read that byte and so on. The program could manipulate a pointer in order to do lseeks reads writes etcetera, but just operate on main memory. Therefore, this would actually simplified things from the perspective of the need for copying there is no longer any need for copying data from a buffer into a buffer of the program.

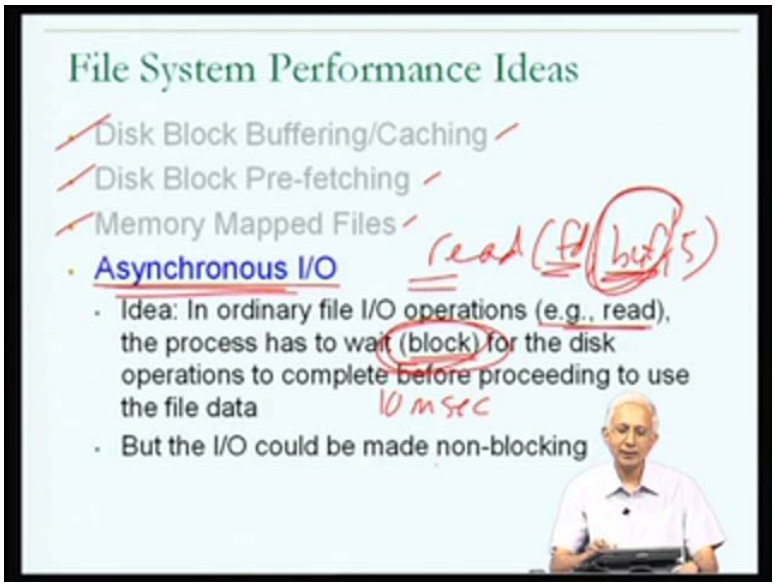
The program is directly accessing the contents of the file from its virtual address space, because of the management of the virtual address space. Now, what happens if a particular part of the file has not been copied into memory because of the equivalent of a page fault? Then, this would be have to be handle much like a page fault. And the copy of the disk may have the copy of the disk block may have to be transferred from the disk into the disk block buffer cache. And subsequently handle; it gives to handle the page fault on that part of the virtual address page corresponding to the mapping of the file into the disk or into the main memory.

So, they could be something equivalent to the page fault, but that could be handled very much like the page fault is handle for any part of the virtual address space of the process. Therefore, there is substantial benefit from using a memory mapped file, if one thinks about the data copying and other aspects of manipulation of the contents of a file. Now, the question which will be arise in your memory is this facility which is available typically on provided by operating systems? The answer is yes, it is provided by every operating system that we have talked about in this course. And in Linux and Unix systems, the corresponding system call is referred to as mmap. And in the mmap system call one must mention which file one is interested in mapping, that is why there is a file descriptor parameter in the mmapped system call, mmapped stands for memory map.

So, one specifies which file one is interested in mapping. One specifies where in memory one is interested in doing the mapping too, in other words the start address and various other parameters. So, the facility is available, the next question that you will ask is or come I have in heard about it before, is it because programs typically do not use mmap? Now, to answer that question, let me just point out that in many of the systems that you use, it may well be the case that the programs that you refer to as cat or c p may benefit from using mmap, rather than using the traditional mechanisms fopen or open read write etcetera.

And that if you actually check to see what mechanism is used by cat or by c p, c p is the program which is used to copy one file into another file. You may well find that these programs are using mmap, because that is the most efficient mechanism for dealing with the kinds of file access patterns that they are handling. And therefore, mmap may be one of the better kept secrets, but it is a facility which is provided widely provided and certainly a value to many kinds of programs.

(Refer Slide Time: 36:00)


File System Performance Ideas

- ~~Disk Block Buffering/Caching~~
- ~~Disk Block Pre-fetching~~
- ~~Memory Mapped Files~~
- Asynchronous I/O
  - Idea: In ordinary file I/O operations (e.g., read), the process has to wait (block) for the disk operations to complete before proceeding to use the file data
  - But the I/O could be made non-blocking

read (fd, buf, 5)
10 msec

Which is why I have included it as a very important performance idea from the perspective of file systems, it is something which we should be aware of you can bet that, the operating systems that you deal with provide disk block buffer caching. And they may be doing good deal of pre-fetching from which your programs that do sequential access will benefit. But most certainly one could think about writing programs, which do memory mapped files in order to get additional benefit. Since, this facility is available on the computer systems that you deal with. Now, there is one other aspect relating to file systems that I do want to talk about in terms of performance and that relates to something which is called asynchronous I/O. unfortunately we have not come across this word before.

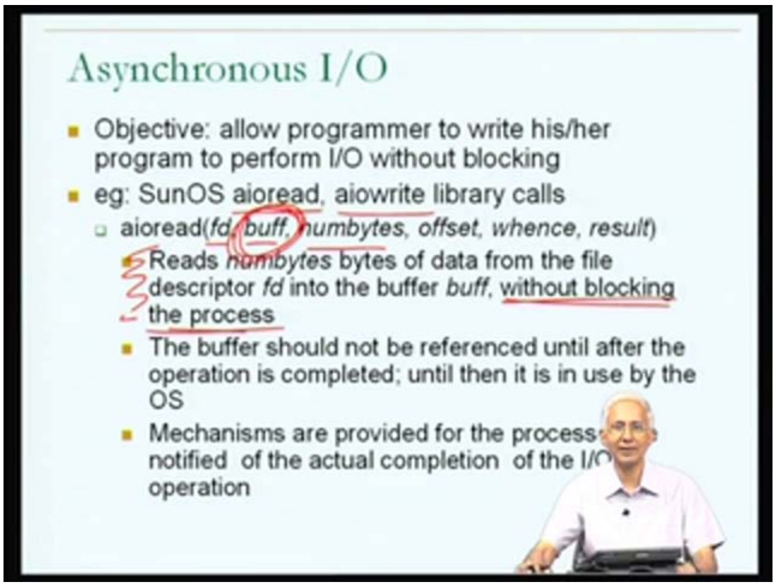
So, I will just have to get into it without any guess any attempt to guess what this might be? Which is what we did with each of the previous terms? Now, the idea in asynchronous I/O is based on the observation, then in ordinary file I/O the kind of file I/O that we were talking about up to now, such as doing a read or a write. We assume that the process would have to wait or get blocked for the disk operation to complete before it could proceed to use the data that had been read. So for example, you think about read, the default for read was that you read from a particular file and the data will be available in a buffer, but that could take ten milliseconds.

Depending on whether in the worst case it could take ten milliseconds if a file I/O disk I/O operation actually had to take place. And in that particular situation it would be necessary form of the program which did the read to not try to access the buffer for that period of time. Or better still the operating system noting that this was an operation which might take 10 milliseconds would block the processes. If would no longer be the running process what would be a ready process, but the net effect is going to be that the amount of time that my program takes to execute. If I look at the elapse time rather than the virtual c p u time would increase.

And that if I had more control over what happens in I/O then I could actually cause my program to use to do other activities while the buffer was being filled with a data that I need. And that I would like to have that kind of control so, this is exactly what asynchronous I/O provides. It provides a mechanism through which the user can write programs, which can do other activities while something like a file I/O is completing. So, the idea of what you might call non-blocking I/O, the disk I/O operation like a read. In the traditional kind of I/O, that we talked about may be a situation where the operating system will put this process into the waiting queue. But if one has special I/O calls through which one can let the operating system know that one has other things that the program can do.

Then they could be this notation of non-blocking I/O and this is what I will refer to as asynchronous I/O.

(Refer Slide Time: 39:19)



Asynchronous I/O

- Objective: allow programmer to write his/her program to perform I/O without blocking
- eg: SunOS aioread, aiowrite library calls
  - aioread(*fd*, *buff*, *numbytes*, *offset*, *whence*, *result*)
    - Reads *numbytes* bytes of data from the file descriptor *fd* into the buffer *buff*, without blocking the process
    - The buffer should not be referenced until after the operation is completed; until then it is in use by the OS
    - Mechanisms are provided for the process to be notified of the actual completion of the I/O operation

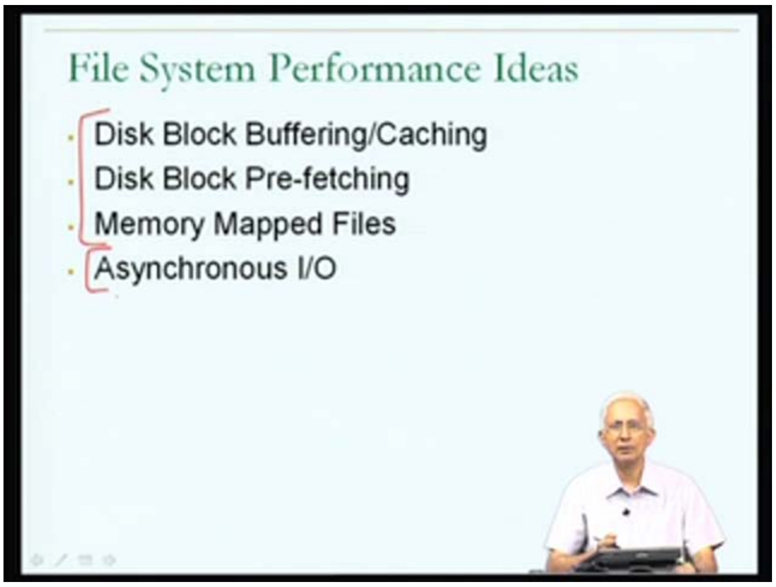
So, in general the objective of an asynchronous I/O mechanism would be to allow the programmer to write his or her program to perform I/O without blocking. In other words to assume that there are other activities which could be done while the disk I/O operation is completing and this is provided by many operating systems through a special set of calls. For example in the sun operating system, there are the asynchronous I/O read and the asynchronous I/O write calls they are known as a I/O read and a I/O write. And they look very much like the system calls that we know before, they require of file descriptor a buffer a num bytes etcetera. And the property that they have is, they cause a number of bytes of data to be read from a file descriptor into a buffer without blocking the process.

In other words, the process which does in asynchronous I/O call will continue to be the running process that is the guarantee that the operating system unless the process is preempted. If the process could be preempted, because at the end of it time class has arrived, but the activity of doing a file I/O operation by itself will not cause the process to get blocked if one uses an asynchronous I/O call. Now, of course, there is a requirement that the user; the program should not attempt to access the data out of buffer until the I/O operation has completed. Because the data will not be available in the buffer until the I/O operation has been completed.

And therefore, they should be some mechanism by which the program can be written so that after they called a I/O read. It does some other useful work, but it has to be able to do enough other useful work until the buffer has been successfully filled. And the program should have some mechanism by which it can check whether the buffer has been filled with a data or not. And therefore, some mechanism must be provided for the process to able to be notified of the actual completion of the I/O operation and put together this provides the user with sufficient additional control. So that, the fact that an I/O operation may take even 10 milliseconds can be overcome to do other useful activity that may have to be done by the program instead of just yielding the c p u and getting blocked.

So, asynchronous I/O was another much more sophisticated kind of a feature, it will require a complete rethink on how you write a program. Because you will have to write the program to do the asynchronous I/O and then to start doing activities which do not relate to the data which is going to be provided from the file as a result of the completion of the I/O.

(Refer Slide Time: 41:40)



File System Performance Ideas

- Disk Block Buffering/Caching
- Disk Block Pre-fetching
- Memory Mapped Files
- Asynchronous I/O

Now, in short then we have seen four interesting and very aggressive kinds of ideas. Some of which will be present in pretty much any operating system, some of which one would have to look to see whether the operating system supports. But all of which can be use by a programmer to aggressively cause the programs that he or she writes to more effectively use files in a performance effective fashion. And we stop here today with our discussion of file system performance ideas. Thank you.