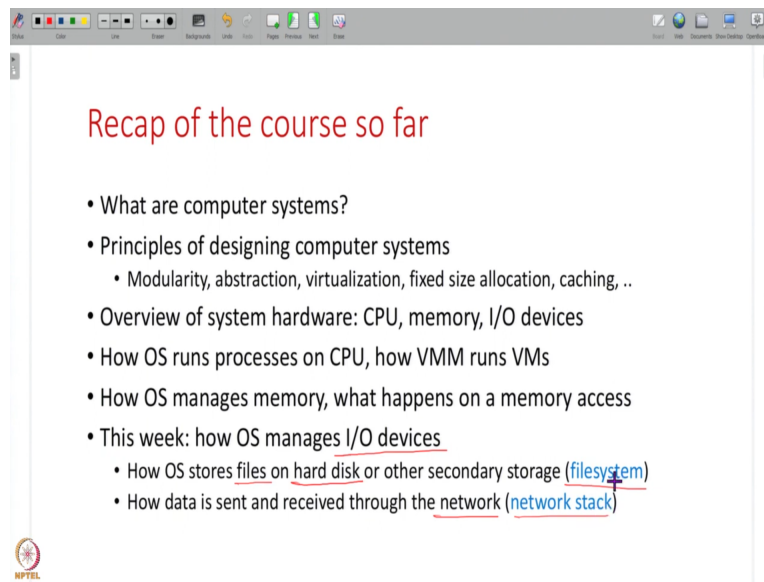


Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay
Lecture 16
Filesystem Data structures

Hello, everyone. Welcome to the 16th lecture in the course Design and Engineering of Computer Systems. So in this week, we are going to study a little bit more about the I/O subsystem in operating systems. So let us get started. So this is a recap of what we have seen in the course so far.

(Refer Slide Time: 1:53)



So we have studied the basics of what are computer systems? What are some of the common principles of designing computer systems? If you remember, from the first week, we had studied principles like abstraction, virtualization, fixed size allocation, caching, and you would have seen these principles being used over and over again in the past few weeks.

We have started this course with an overview of the hardware, CPU, memory, and I/O devices and then we moved on to the system software layer, which is we have seen how the OS runs processes and the similar concept of virtualizing the CPU for a process was used to run virtual machines.

And then we have studied a little bit about how memory is accessed and managed in an operating system. And this week, we are going to complete our discussion of operating systems by understanding how the OS manages I/O devices. In particular, we are going to focus on two types of I/O devices, mainly, that is the hard disk or secondary storage, which stores your files and the network card or the network interface which lets you communicate with other machines.

So these are the two main important subsystems in an operating system when pertaining to dealing with I/O devices. So the part that deals with files is called the filesystem and the part of the OS that manages the network is called the network stack. And in this week, we are going to study these two in a little bit more detail.

(Refer Slide Time: 02:08)

The slide is titled "Recap of I/O devices" in red. It contains a bulleted list of points about I/O devices, device controllers, and DMA. Handwritten red annotations include: "CPU" with an arrow pointing to the "Controller" box; "OS" with an arrow pointing to the "Command" label; "disk" with an arrow pointing to the "disk" label; "RAM" with an arrow pointing to the "RAM" label; and "Interrupt" with an arrow pointing to the "Interrupt" label. The annotations also show a flow from "CPU" to "Controller" to "I/O" to "Command" to "disk" to "RAM" to "Interrupt" back to "Controller".

Recap of I/O devices

- I/O devices: expose block storage (hard disk) or stream of bytes (network card, keyboard, ...)
- **Device controller**: microcontroller that manages I/O device
 - Exposes various registers: command, status, data
 - **Device drivers** reads/writes these registers to communicate with device
- Example: reading data from disk
 - Device driver gives command to read via command register
 - Device controller executes disk read on disk hardware, transfers data into main memory via DMA, raises interrupt
 - Device driver handles interrupt, processes received data
- **DMA** reduces I/O overheads, especially for high speed devices (disks, network cards)

So, let us get started. So this is a brief recap of what are I/O devices and how they work that we have seen in the first week of the course. So an I/O device is any device that is other than the CPU and your main memory that is there on your computer, like a hard disk that stores data in blocks or other devices like network cards, keyboard, which basically generate a stream of bytes, or even your monitor, which is an output device and so on. All of these I/O devices, and you all are familiar with this.

So every I/O device is usually managed by a device controller, which is like a specialized microcontroller that focuses on managing the I/O device. So there is the I/O device and there is

the device controller, which exposes, or which has a bunch of registers. And the CPU or the device driver in the operating system will access these registers, read and write these registers of the device controller in order to get some I/O done using the I/O device.

So we have seen that some common registers are the command register, which is used to give a command, the device driver will write a command into the command register. The status register indicates what is the status of the I/O device. The data register is used to read and write data from the I/O device and so on. And device drivers communicate with the device through these registers.

So we have seen a simple example of how you read the data from a disk. So the device driver that is running inside the operating system will give a command to the disk asking it to read, say, some block of data on the disk. And once the command is given, of course, the operating system will, you know, whichever process gave the command it will block, it will be context switched out and the OS will run some other process simply because the disk takes a really long time in order to finish this operation as compared to the operating systems or the CPU speed. So therefore, while the disk is executing this command, the OS will go run some other process.

And at this point of time, the disk is executing the command. And when the data is ready to read, whatever data has been asked to read, when that is ready, the device will transfer it into RAM, it will perform a DMA, a direct memory access and transfer the data into main memory.

And then it will raise an interrupt. The trap instruction will run. Then the OS, the device driver will come, handle the interrupt, look at the data, process it further, you know, give it to the process, unblock it, all of that will be done. So we have seen all of this before. And the main reason why we use DMA is that it reduces the overhead instead of the device driver copying data from the some register of the device controller, the device is directly putting the data in RAM which makes life easy for the interrupt handling part.

And this is especially important for high-speed devices like disks and network cards and so on. So in the next few lectures, whatever we study, they are all based on DMA. So this is a recap of what we have seen before in the first week of the course with respect to I/O devices.

(Refer Slide Time: 05:31)

Filesystems

- **File:** persistent storage of user data on secondary storage (hard disks, ..)
 - Traditional hard disk stores file data in fixed size blocks
- **Filesystem:** OS code that manages files on disk
 - User programs invokes system calls to access files: open, read, write
 - OS implements system calls and performs operations on underlying disk data
 - OS system call implementation accesses blocks on device via device driver
 - OS also manages disk buffer cache which caches recently accessed disk blocks
- A filesystem is a specific way of storing and organizing file data on disk
 - Many ways to organize data, many filesystems exist (e.g., ext3, ext4, FAT)
- This lecture: Data structures used in a simple filesystem
- Next lecture: Implementation of filesystem-related system calls

Now let us begin our discussion of filesystems. So we have seen the notion of a file before, a file is nothing but a stream of bytes that are persistently stored on a secondary storage device like a hard disk. You all know what the file is. You read and write files, access files all the time on your computers.

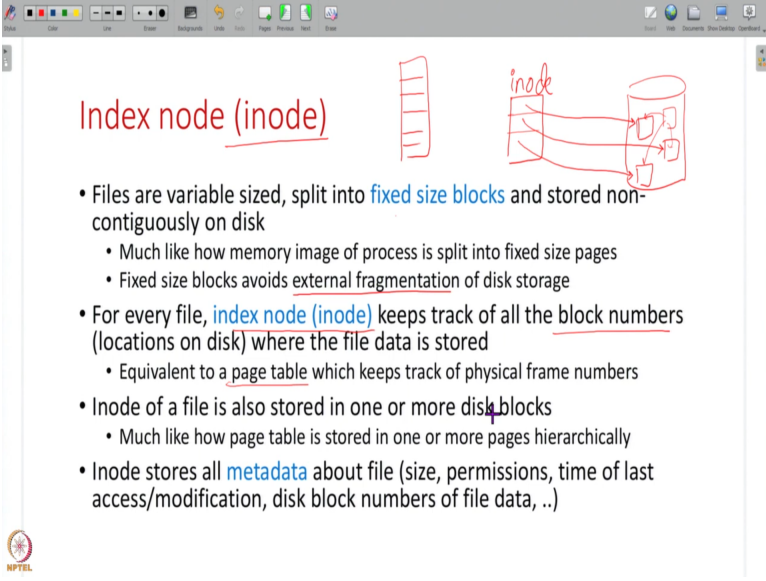
And traditional hard disks split a file into some fixed size blocks and store it on disk. It is not stored continuously, but into, it is split into fix size blocks and stored. And the operating system, what does the operating system have to do with files? The OS does many things. First is of course the user program is given a bunch of system calls. The OS exposes an API of system calls to user programs. And then the OS actually implements these system calls. You know, your system calls to open a file, read a file, write a file, all of those are implemented by the operating system and the part of the operating system that is called the filesystem.

And as you implement these system calls, you will have to access the underlying disk, the device you have to access, and this access has done via device driver. So this accessing the actual device is done via the device driver. And also, in addition to all of these pieces, we have also seen that the OS has maintains a cache, a disk buffer cache off the recently accessed disk blocks, whatever you have read from disk via the device driver, you will cache them for some more time.

So all of these things, the system call implementations, the device driver, the disk buffer, cache all of these together are used by the OS to provide the user, the abstraction of files, and all of the code together is called the filesystem. And every filesystem has a specific way of organizing data on disk, you know, how to split a file into different blocks and so on. There are different ways of doing it, there is not just one way of doing it, which is why there are many different filesystems in use in modern operating systems.

So in this lecture, we are not going to go into details about any particular filesystem, but we are going to study the general principles that hold across any filesystem. So in this lecture, we will begin with understanding what are the data structures that are used in a simple file system. And in the next lecture, we are going to study the implementation of this system calls. How do you actually do open, read, write using these data structures.

(Refer Slide Time: 08:10)



The slide is titled "Index node (inode)" in red. To the right of the title is a diagram showing a vertical stack of five rectangular blocks. To the right of that is a cylinder representing a disk, with several small squares on its surface. Red lines connect the blocks to the squares on the disk. The word "inode" is written in red above the diagram. Below the title and diagram is a list of bullet points.

- Files are variable sized, split into **fixed size blocks** and stored non-contiguously on disk
 - Much like how memory image of process is split into fixed size pages
 - Fixed size blocks avoids external fragmentation of disk storage
- For every file, **index node (inode)** keeps track of all the block numbers (locations on disk) where the file data is stored
 - Equivalent to a page table which keeps track of physical frame numbers
- Inode of a file is also stored in one or more **disk blocks**
 - Much like how page table is stored in one or more pages hierarchically
- Inode stores all **metadata** about file (size, permissions, time of last access/modification, disk block numbers of file data, ..)

So let us begin the first and the most important data structure in the filesystem, in modern filesystems is what is called the inode or the index node. So what is the inode? A file, a variable sized file is stored in, is split into fix sized blocks and stored on the disk, that is, if you have a large file like this, you will split it into blocks, and then on your disk, you will have one block here, one block here, one block here. We will not store the file contiguously, but split it into fix size blocks. And this is a principle we have seen before with respect to pages in main memory and so on.

And why do we do this? Because we want to avoid fragmentation, external fragmentation. You do not want variable size things being stored on disk and there being gaps left between two variable size chunks. You do not want to deal with that headache. So we split the disk into fix size blocks, and we allocate storage to a file in terms of these fix size blocks.

So once a file is split into fix size blocks and stored non-contiguously, not in one place, but all over the disk, you will need some way to keep track of where all a file is located on disk. Just like how the memory image of a process is split in across many pages in memory and you need a page table to keep track of it.

Similarly, for a file also, every file, there is a index node or an inode, which actually remembers the location of all the blocks, where a file is stored. The first few bytes of a file is stored in this block number, the next few bytes in this block number, the next few bytes in this block number.

So all of these block numbers where a file data is stored, all of those are kept track of, all of those block numbers are stored in the inode or the index node of a file, so this index node of a file is also stored on disk, just like how your file data is split into blocks and stored on disk, similarly, the inode is also stored on disk someway, and this inode has the block numbers of all the blocks of a file.

And this inode need not fit in one disk block, again, it can be split into multiple disk blocks. You will have a hierarchical structure that we will see in a little bit. So conceptually, it is similar to a page table of where you do fix size allocation, non-contiguous allocation and keep track of all that information in an index block.

So the inode stores all metadata about a file. Not just the location of these data blocks, the disk block numbers of the file is one type of metadata about a file. So understand the difference between the data of a file which is the actual contents of the file and the metadata of a file, which is information about the file, which includes disk block numbers, various other things like what is the file size, permissions, who can read, who can write, when was the last time, the file was accessed, modified, all of this constitutes the metadata of a file, and the inode stores all of that information also. And this inode is also stored on the disk along with the data blocks of a file.

(Refer Slide Time: 11:34)

Structure of inode

- A file is uniquely identified by its inode number on disk
 - An inode number uniquely identifies disk block containing inode on disk
- How are block numbers of file data blocks stored in inode?
 - All block numbers of a file may not fit in one inode disk block
- Hierarchical method of storing block numbers in inode
 - Inode contains the block numbers of first few blocks of a file (direct blocks)
 - If direct blocks are full, inode contains block number of single indirect block, which contains block numbers of next few blocks of a file
 - If single indirect block is full, inode contains block number of double indirect block, which contains block numbers of more single indirect blocks
 - Triple indirect block can also be used for large files
- Not a symmetric hierarchical structure like page table
 - Most files are small, so first few block numbers of a file are made available easily without accessing multiple levels of inode
- Accessing a file from disk may require multiple disk accesses for inode

So let us understand the structure of the inode in a little bit more detail. So every inode in a filesystem has a unique number, the inode number, it is like say the roll number of a student, which uniquely identifies the inode on disk, and therefore, from the inode, you can uniquely identify your file. So given a file, if you know its inode number, you can locate the inode on disk.

And once you locate the inode, you know where all the data blocks are there of the file and you can get all the information about a file by knowing inode number of the file. So this is a very important piece of information about the file.

Now, inside the inode, how do you store all the block numbers of a file? A file can have many blocks, typical block sizes on a traditional hard disk is say 512 bytes. So if your file is larger than that, you will, the file will be stored in multiple data blocks. And how do you store all of these data blocks in an inode, we are going to understand now. So note that all of the data blocks of a file may not even fit in one inode. You might have to use multiple different blocks. They may not fit in one inode disk block.

So the way it is done is again, the same idea of hierarchically storing, like we have seen in page table, so similar idea is used. So the typical inode looks like this. The first few block numbers of a file, the first few blocks of a file, their numbers are directly stored in the inode, that is the inode we will have pointers, direct pointers to the first few blocks of a file. And those blocks are called direct blocks.

Why are they called direct blocks? Because their block number is directly there in the inode. You go to the inode, you the first block of a file, block number 42, you know, that you can immediately access the first block of the file. So this way the, inode will have a first few direct blocks.

And then of course you cannot store large number of things, at some point, space in the inode will run out. Then once all the direct blocks are full, once your file size exceeds these direct blocks, then what you will do is you will create an indirect block, which in turn, has the block numbers of the file, and this indirect block number, that will be stored in the inode.

So there will be a single indirect block, which has the numbers of the data blocks of the file and the number of the single indirect block is stored in the inode. So without occupying much space, you are able to store many more block numbers using a single indirect block. Then what if this is also full? What if your file is bigger than this?

Then you will have a double indirect block, which is your inode will have a block number of a double indirect block and this double indirect block we will have the block numbers of several single indirect blocks. And these single indirect blocks will have the actual block numbers of the file, that is, you have a two level hierarchy.

The block numbers of the file are stored in these single indirect blocks. These block numbers are stored in the double indirect block. And this block number of the double indirect block is stored in the inode of the file. Similarly, you can keep doing this. You can have a triple indirect block and so on.

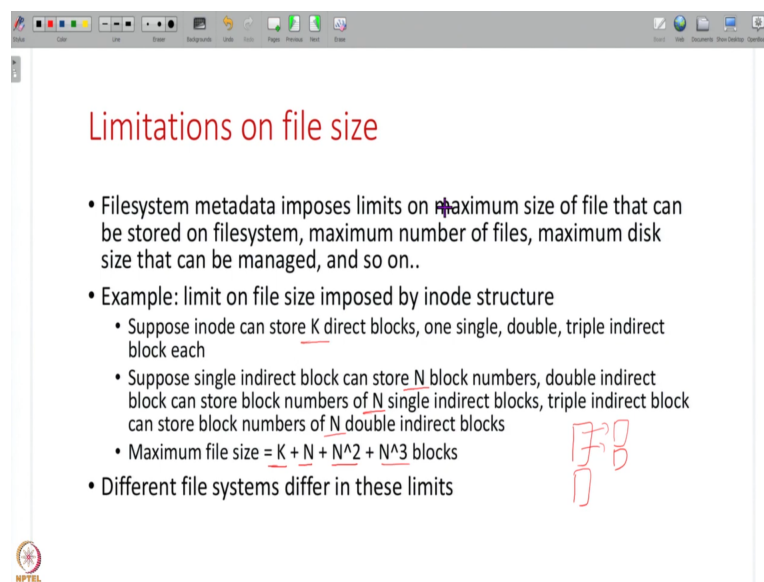
So normally, filesystem stop around a triple indirect block, because you do not need to store files larger than that. So, note that this is not the symmetrically hierarchical structure, like a page table, a page table, everything was two levels, in the two level page table to access any virtual address, any page you had to go through two levels, but here it is not like that.

The first few block numbers of a file, you can get by directly from the inode. Then the next few block numbers, you will have to do one hop, the next few block numbers you will have to do two hops, then three hops and so on. So why is this hierarchical structure? Because accessing a file may require multiple disk accesses and it is very slow.

And therefore, if most files are just small, you want to avoid so many disk access as you want to avoid multiple disk accesses even for smaller. So small files, I will try to fit them into the direct blocks as much as possible. Only for larger files, the overhead will increase. So that is the reason why this inode is somewhat of a asymmetric hierarchical structure, unlike a page table.

And as with any multi-level indexing structure, accessing a file now requires multiple disk accesses. So every time you want to read some data block of a file, first you have to get the inode, from there, may be read multiple other blocks, and then finally know the block number that you want of the file, and then read that disk block. So your overhead has increased due to this inode structure.

(Refer Slide Time: 16:25)



Limitations on file size

- Filesystem metadata imposes limits on maximum size of file that can be stored on filesystem, maximum number of files, maximum disk size that can be managed, and so on..
- Example: limit on file size imposed by inode structure
 - Suppose inode can store K direct blocks, one single, double, triple indirect block each
 - Suppose single indirect block can store N block numbers, double indirect block can store block numbers of N single indirect blocks, triple indirect block can store block numbers of N double indirect blocks
 - Maximum file size = $K + N + N^2 + N^3$ blocks
- Different file systems differ in these limits

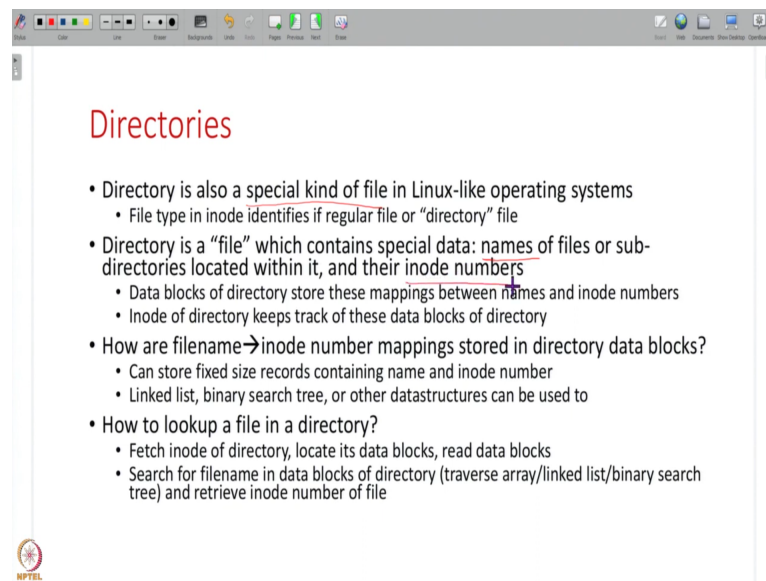
Handwritten red notes and diagram: To the right of the text, there are handwritten red notes. A vertical line is drawn, and to its right, the numbers 1, 2, and 3 are written vertically, corresponding to the levels of indirect blocks mentioned in the text.

And this inode also imposes a certain limit on your maximum file size. As you can see, there is only so many data block numbers of a file you can store in the inode and that limits your file size. For example, if your inode has K direct blocks and a single indirect block that can store N other block numbers and a double indirect block that can store N single indirect blocks, a triple indirect block and so on. Then what is a maximum file size that you can keep track of in your inode? You can remember the first K direct blocks, then you can remember N indirect blocks, that is the block numbers, which are stored in the single indirect block.

Then you have, you know, N single indirect block, each of which has N block number. So therefore, from the double indirect block, you can store N^2 block numbers. Then from the triple indirect block, you can store N^3 block numbers. So this will be the maximum size of the file you can keep track of. Once your file size exceeds these many blocks, then there is no more space in your inode available in order to store the block number of your extra data in the file beyond these many blocks. If you need more blocks, there is no place to store the block number in your inode, therefore you cannot keep track of the file anymore. Therefore, there is a limit on the file size.

Similarly, there could be limits on, you know, depending on how many inodes you have? How many files you can have in your system? What is the size of the disk that you can manage? All of these based on how your filesystem is structured? How these metadata data structures are structured? Based on that, there is certain limits on various file size and disk size and so on in any filesystem, and different filesystems have different limits, because their data structures are different. So next we come to directories.

(Refer Slide Time: 18:23)



So, what is a directory? You all know what is a directory, which is a collection of files. But in Linux-like operating system, their directory is also treated like a special kind of file. It is not any separate entity, it is also a file. Just that it is a file which contains information about other files. It is a special kind of file, which contains information like the names of all the files in that directory and their inode numbers.

So the directory, if you look at, if you treat the directory as sort of a text file or something as a regular file, what you will find inside the directory is a file name and an inode number. You know, file name inode number, file name inode number, subdirectory inode number. And this is what you will find inside a directory. If you were to read it like a regular file.

So that, therefore the directory is also a file. It will also have an inode number, it will also have an inode and all these contents of the directory will also be split into blocks, spread all over the disk and the inode of the directory we will keep track of all of these data blocks of a directory, so it is treated like a file. And the file type in the inode will indicate, is it a regular file? Is it a directory kind of a special file? All of that is indicated. And the inode of a directory keeps track of all these data blocks of a directory.

And how do you store these file names to inode number mappings in the directory data blocks? You can store it in any format. You can say, I will have fix size records. You know, there is a fix

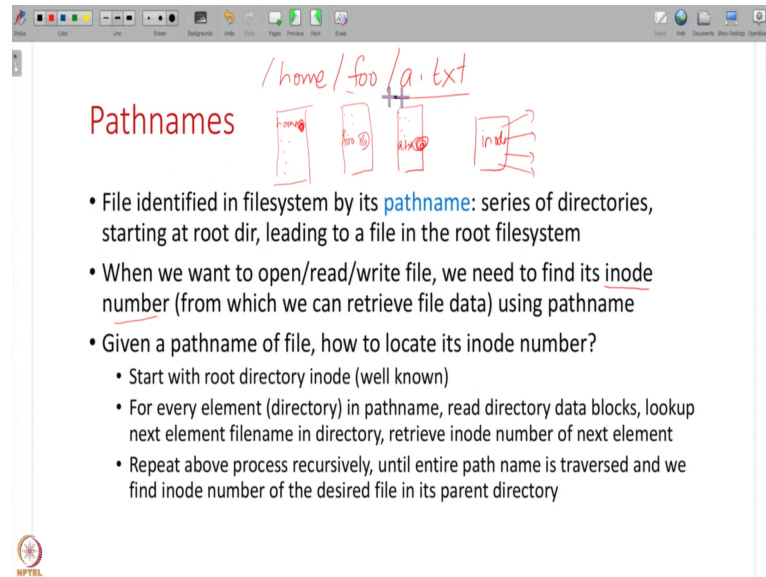
size record that has the file name and inode number, another fix size, another fix size. You can store it as fix size records, you can store it as a linked list, you can store it as a binary search tree sorted by the, alphabetically sorted by the file name. There are many different ways in which you can store this file name to inode number mapping in a directory.

So if you want to find a file in a directory, how do you do that? So there is the inode of the directory itself that has information about all these data blocks of the directory. So you will first fetch the inode of a directory. And inside that inode of the directory, you have all these data block numbers of the directory, you will locate the data blocks, you will read the data blocks of the directory. And inside those data blocks, you will find all of these records of a file name, its inode number, file name, its inode number, all of that information you will find inside the data blocks of the directory, and then you will search through this information for the particular file you want.

And once you find your file, you know its inode number, and from the inode number of a file, then you know the information about the data blocks of the file itself and then you can proceed to access the file. In this way, the directory helps you locate the information about a particular file. And note that the search for the file name in a directory will depend on how the directory is storing these records. Is it storing them as fix sized records, link list, search tree. So depending on that, you are searching for this file, the logic will be different inside a directory.

But the basic concept is as follows, a directory is nothing but a special file that has mappings between the file names and their inode numbers so that you can locate these inodes of a file when you need to. And this directory, this is the data blocks of a directory. Every directory also has an inode, which has pointers to these data blocks. And this directory's inode, this number will be there in its parent directory. The parent directory will keep track of the inode number of sub-directory.

(Refer Slide Time: 22:21)



Pathnames

/home/foo/a.txt

- File identified in filesystem by its **pathname**: series of directories, starting at root dir, leading to a file in the root filesystem
- When we want to open/read/write file, we need to find its inode number (from which we can retrieve file data) using pathname
- Given a pathname of file, how to locate its inode number?
 - Start with root directory inode (well known)
 - For every element (directory) in pathname, read directory data blocks, lookup next element filename in directory, retrieve inode number of next element
 - Repeat above process recursively, until entire path name is traversed and we find inode number of the desired file in its parent directory

So now next, let us understand path names. So what are path names? You all know what path names are. In any file system, there is some path name like, you know, slash home, slash foo, slash a dot text, this is sort of in a, in the directory tree, in the root filesystem, starting with the root directory. This pathname helps you locate a particular file. A file is identified by its pathname. Now every time you want to open or access a file in anyway, you need to know its pathname so that you can find out more information about the file.

So if you know the pathname, how do you find out the inode number of the file? It is as follows. You will first start with the root directory. You will know the inode number of the root directory. From that, you will get all the data blocks of the root directory.

And in the data blocks of the root directory, you have information like this home, sub-directory, some other sub-directory, what their inode numbers are. You know, those inode numbers. From this inode number of the home directory, you get its data blocks, and inside the home directory, there are many other directories, they are like the foo directory, and you will get the inode number of the foo directory.

And in the foo directory, you have many files like, you know, a dot text, and you know its inode number. And from this inode number, you will get the inode of a dot text. And in that inode, you

have the data block numbers of a dot text you can access the file. So this is called recursively traversing the pathname of a file in order to get to the information in the file.

So you start with the root, and for every element, you will read the directory's data blocks, look up the next element in the pathname, retrieve its inode number, and again repeat, again read the next level, its inode, its data blocks, and the next level you find out, its inode, its data blocks, you keep repeating this process until you reach the end of your pathname.

At which point, you have the number of the file that you want from that inode number, you will get information about all the data blocks of a file. So this is how you traverse a path name to find out more information about a file, like its inode number and its data blocks.

(Refer Slide Time: 24:45)

Disk layout of filesystem

- What all does hard disk have?
 - Data blocks of files and directories
 - Inode (metadata) blocks of files and directories
 - Information about which data/metadata blocks are free and which are occupied
 - Overall master plan of disk is stored in the first block: superblock
- Free space management: how to know which blocks are free?
 - Free list: superblock contains disk block number of first free block, which contains block number of next free block, and so on..
 - Bitmap: few blocks on disk contain bitmaps, one bit of information about each disk block, whether free or not
- All this layout is done when a hard disk is "formatted" with a filesystem
 - Different filesystems will have different layouts, formatted differently
 - Maximum number of files, maximum disk size etc. depend on this format

So now any filesystem has a certain layout on disk. There are all of these things, you know, inodes and directories and everything, and all of these are laid out, organized on disk in a certain way. So every hard disk, if you look at it, it will have, you know, a large number of data blocks. Which store, you know, the data of files as well as directories, you know, even directories need data blocks to keep track of file name to inode number mapping.

So you have a bunch of data blocks, and then you have a bunch of inode blocks, and then you need to have some information about which of these data blocks are free, which are occupied,

which are not, all of that information is also maintained, you know, in the form of, say, which, like a free list, which blocks are free or some kind of a bitmap.

So what is a bitmap? You have one bit indicating 0 or 1 is the block on disk free or not. You have that information, that is called a bitmap. So you have all of these things stored on the hard disk. And then there is one block. The first block of the filesystem is called the superblock, which basically has information of this layout itself.

The superblock will say, okay, the first few blocks contain the bitmaps, the next few blocks contain the inodes, that next few blocks contain the data blocks, all of that, this organization, this information is stored in the superblock. So anytime you want to understand how a filesystem is organized on disk, you start with the superblock, read information about all the other organization in the superblock, and then you can access the rest of the filesystem.

And how is free space managed? We have seen, one way is the free list, which is, you remember, you store the free block numbers as a list and the first free block number you remember in say your super block or somewhere, and in this first free block, you store the block number of the next free block, in this free block, you store the block number of the next free block and so on. You can maintain like a free lists like this.

Or you can store a bitmap as we have discussed above, you maintain one bit of information about each disk block, that will tell you is that disk free or is it in use by some file or some directory or something? So in this way, you will manage the free space on a disk and this free space information, these bit maps are also stored on the hard disk.

And when you format a hard disk, if you say, you know, you must have heard the term formatting a hard disk, what does it mean? It simply means laying out this information on the disk in a specific format. And different filesystems will have different layouts, different data structures for all of these things. And therefore, the formatting also will be different.

(Refer Slide Time: 27:43)

The slide is titled "In-memory data structures" in red. Above the title is a diagram showing a vertical array of boxes labeled "fd" (file descriptor) with an arrow labeled "open" pointing to it. Another arrow labeled "offset" points from the "fd" array to a second vertical array of boxes labeled "inode". A third arrow labeled "inode" points from the "inode" array to a separate structure on the right. Below the title is a bulleted list of in-memory data structures.

- When a file is opened, in-memory inode is cached from on-disk copy
 - Quick access of file data block numbers as long as file is in use
- Open file table: data structure used by kernel to keep track of open files
 - One open file table entry created for every open system call
 - Contains pointer to in-memory inode and other information about open file (e.g., offset at which the file is being read/written)
- File descriptor array: per-process array of open file table entries for files that are opened by the process (part of PCB)
 - When you open a file, open file table entry is created, its pointer is stored in file descriptor array and index in array is returned as file descriptor/handle
 - OS can locate file inode for reading/writing using file descriptor
- Disk buffer cache: LRU cache for recently read blocks from disk
- Next lecture: how all these data structures are used in implementing system calls

So next is the in-memory data structures. So all of this is on disk. For a filesystem on disk, you have data blocks, you have inodes, you have bitmaps, you have a superblock, all of this is on the hard disk pertaining to a filesystem. Now a filesystem also in memory, when the OS running, when the system is running, in memory also, it has a few data structures to keep track of. And what are they? When a file is open and when it is being used to read and write, you will basically take the inode from disk and you will keep a copy of it in-memory in a cache, that is called the in-memory inode.

Why is this done? Because when you are reading, writing a file, you need to know which is the first block number, second block number you need to know a lot of information about the file. Every time going to the disk to access the inode is a little bit suboptimal. Therefore, to speed things up, you will bring the inode into memory for the duration that the file is opened and in use, that is the in-memory inode.

Then you have a data structure called the open file table which is used to keep track of all the open files. So this is a table, it has one entry for every file that is opened, and this entry contains, you know, say the pointer to the in-memory, the inode number or various other pieces of information. The other piece of information is the offset,

Remember that in a previous lecture, we have discussed, when you read a file, you read it as a stream of bytes, if you first read 64 bytes, the next time you read, you will get the next 64 bytes. So you have to remember the offset in a stream of bytes until which point have you read. So that offset is also kept track of in the open file table. So this open file table has the pointer to the in-memory inode, things like offset and a few other things. All of this information is stored in the open file table.

And in addition to this open file table, you have what is called the file descriptor array. So this open file table is a global data structure. There is one for all processes in the system, whereas this file descriptor array is per-process array of information about the files that are open for this particular process.

And this is part of the PCB and the PCB is where you keep track of all information about a process, the file descriptor array is also part of the PCB, and this file descriptor array for every file that is open, they inform, the pointer to those open file table entry will be stored in this file descriptor array.

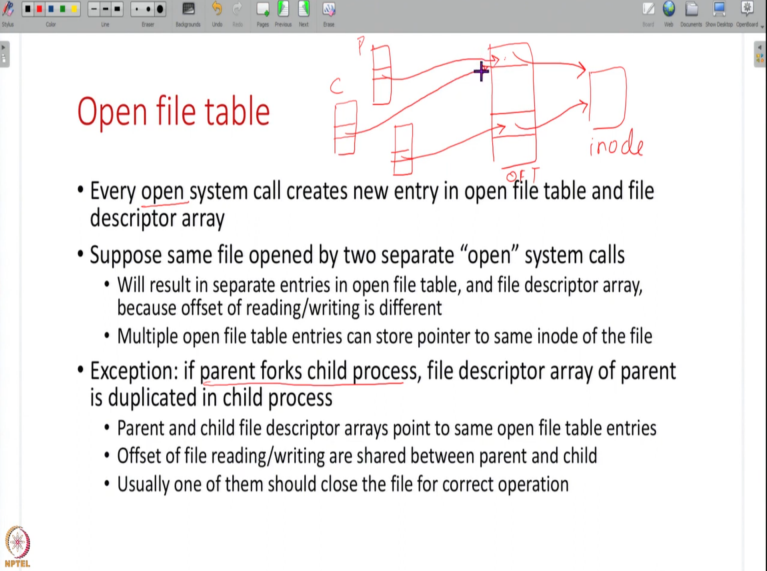
So that this index in this file descriptor array, this is what is returned to you when you open a file. So when you open a file, the inode is brought into memory, an open file table entry is created and the pointer to this open file table entry is stored in the file descriptor area and which index is at, the 0, 1, 2, whichever index in the file descriptor array you have stored this information, that is returned to you as a file descriptor or a file handle.

And in the future, for any operation on the file, you provide this handle, and from this handle, you follow the pointers, you will reach the inode of the file, you have all information about the file readily available for you. So in the next lecture, we are going to see how all of these data structures are used when we open a file, read file, write a file and so on. We will come to that in the next lecture.

And one final in-memory data structure is the disk buffer cache. So all the blocks of a file that you read from disk recently, all of them are cached in a least recently used cache called the disk buffer cache. So these are all the important data structures. Of course, different filesystems, more

complicated, real life filesystems may have a more or complex, different data structures than this, but this is a basic idea for a simple file system.

(Refer Slide Time: 31:41)



Open file table

- Every open system call creates new entry in open file table and file descriptor array
- Suppose same file opened by two separate “open” system calls
 - Will result in separate entries in open file table, and file descriptor array, because offset of reading/writing is different
 - Multiple open file table entries can store pointer to same inode of the file
- Exception: if parent forks child process, file descriptor array of parent is duplicated in child process
 - Parent and child file descriptor arrays point to same open file table entries
 - Offset of file reading/writing are shared between parent and child
 - Usually one of them should close the file for correct operation

And one subtlety I would like to point out about the open file table is that every time you open a file with the open system call, you will have a new entry in the open file table and a new entry in the file descriptor array.

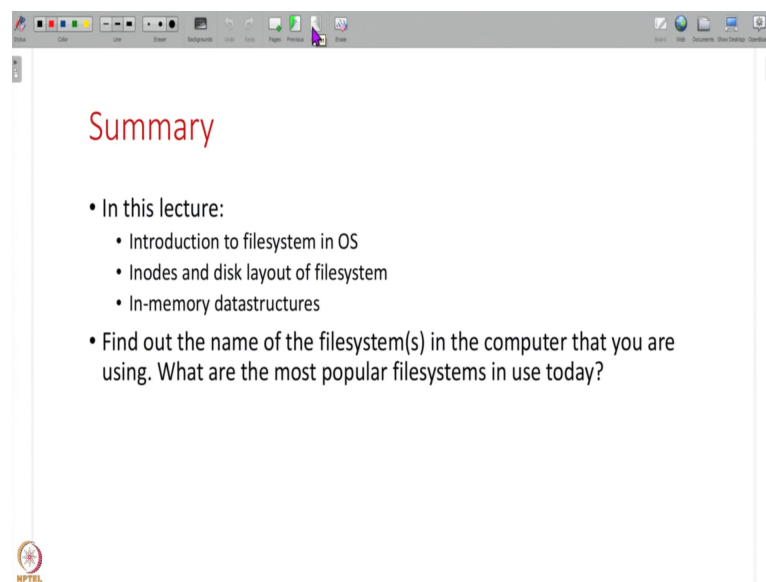
So suppose there is a file that is opened by two different processes. Then the inode of the file will be common, but if process P opens a file, it will create a separate open file table entry which is pointing to this inode. If process Q opens the file, it will also create a separate open file table entry, that is also storing a pointer to the same inode. Similarly, process P will have its own file descriptor array that has a pointer to the open file table entry. Process Q will have its own file descriptor area that has a pointer to its open file table entry.

Why do we have this? Why do we have different open file table entries? This is because the files, when they open, when different processes open a file, the offsets at which they are reading, they are independent. If process P reads 64 bytes, then process Q will start at the beginning, it will not continue where P left off.

Therefore, you need to keep track of different offsets. Therefore you will have different open file table entries. With the exception that when the parent forks a child process, at that point, if this is process P and it has forked the child C, then both this parent and the child share the same offsets of the open files, that is they will both point to the same open file table entry. And when one of them reads or writes, that offset is reflected in the other child.

So this is the only time where different processes will use the same open file table entry. Otherwise, different processes open the same file also, they will have separate open file table entries and they will read the file as independent streams. And in the case of the parent and child, of course, this is a little messy. Therefore, it is usually better if one of them closes their file and the other continues to use it.

(Refer Slide Time: 33:51)



So, that is all for this lecture. In this lecture, we have introduced the concept of what happens in the filesystem of an operating system. We have introduced the datastructures like open file tables, inodes, and so on. In the next lecture, we are going to continue our discussion on how the filesystem, system calls are implemented using these datastructures.

And a small exercise for you is, as I have said, there are many different filesystems in use and modern systems. So take a look at your computer, your desktop or laptop, find out more about what filesystem is it running? What is the name of the filesystem? What are some of its

differentiating characteristics? You might want to just read up a little bit about the different types of filesystems available today.

Thank you all. That is all I have for this lecture. Let us continue this topic in the next lecture.
Thank you.