

Design and Engineering of Computer System
Professor. Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 14
File System and Memory

Hello, everyone, welcome to the fourteenth lecture in the course Design and Engineering of Computer Systems. In this lecture, we are going to continue our discussion on memory management. And we are going to study how memory intersects with the file system. So, let us get started. So far what we have seen in the courses is how user applications store data in the main memory.

(Refer Slide Time: 00:40)

Storing program data

- User applications deal with data (variables, data structures, ..)
 - Global/static variables are in program executable
 - Memory allocated via malloc stored on heap
 - Function variables and arguments stored on stack
 - Can request one or more pages from OS via mmap to store data
- All of these mechanisms do not store data persistently
 - Data in main memory is lost when power is turned off
- How to store program data persistently?
 - Use files on secondary storage (hard disk and other such storage media)
- In this lecture: interaction of files with main memory
 - Next week: more details on filesystem, how file operations are done

The diagram on the right shows a vertical stack of boxes representing memory layout: 'Code + data', 'Stack', 'heap', and 'mmap'. A bracket on the right side of these boxes is labeled 'memory image'. Arrows point from the text in the bullet points to the corresponding parts of the diagram: 'Global/static variables' to 'Code + data', 'Memory allocated via malloc' to 'heap', 'Function variables and arguments' to 'Stack', and 'Can request one or more pages from OS via mmap' to 'mmap'.

So, every application has to deal with some data, whether it is variables, data structures, every program has a lot of data on which it operates. And this data can be stored in many different parts of the memory image of a process. So, we have seen that global and static variables are part of the program executable, and they are part of the code and data section of a program.

So, if this is the memory image of a program, running program or a process, then you have the code plus data that is part of the executable, then you have the stack, heap. And you can also memory map, we have seen this, a few lectures back that the user program can also do the mmap system call to request additional pages to store its own data, in addition to whatever is there on the heap, stack, and so on.

So, global and static variables are in the coordinate data section that is copied from the executable. Malloc memory is on the heap, function variables and arguments, anything, any memory allocated within a function is on the stack. And you also have separate memory mapped regions that the program can use to store data. So, all of these ways of storing data in a user program, all of these do not store the data persistently.

That is when you turn off your computer, all of this data is lost when the process dies, all of this data is lost. So, this is not persistent storage of data. But user programs and applications want to store user data persistently. Your account information in a bank has to be stored persistently and cannot be lost when the computer turns off at the end of the day. So, for persistent storage, we use the disk, we use secondary storage which is persistent or non volatile storage, like the hard disk.

And on it we store the data usually as files. So, files are the abstraction used to store data on secondary storage or hard disk persistently. So, in this lecture, we are not going to go into a lot of detail on how files work and how data is stored in files, that will be the topic for next week. But in this week, since we have already studied a little bit of memory management, we are going to study the intersection of file system and memory.

So, how the file system uses the main memory, that is the part we are going to cover in this lecture. And in the next week, we are going to do a lot more detail on how the file system works.

(Refer Slide Time: 03:20)

File abstraction

Diagram illustrating file system hierarchy:

- Root directory (implied) branches into `home`, `bin`, and `usr`.
- `home` contains a file `foo`.
- `usr` contains a directory `disk2/...`.
- A file `a.txt` is shown within the `usr` directory.

- **File**: sequence of bytes, stored persistently on disk
- **Directory**: container for files and other sub-directories
- **Directory tree**: hierarchy of directories and sub-directories, containing files
 - Root file system: directory tree starting at root ("`/`")
 - Directory trees on disk can be mounted at various locations of root filesystem
- Containers have different root file system view on the same OS
- Steps to access a file
 - Open a file using system call, get a file descriptor
 - File descriptor is a handle to refer to file for read/write
 - Close file when done accessing it

```
fd = open("/home/foo/a.txt")
read(fd, ...)
write(fd, ...)
close(fd)
```

So, with this introduction to this lecture, let us get started on what are files. So, you can think of a file as nothing but, a sequence of bytes. You have a file, say let us call it a dot text, it has a stream of bytes, which is stored persistently on a secondary storage device like the hard disk. And what is a directory? You all are familiar with what are directories, directories are nothing but containers for files when you have multiple files, all of these files together are stored in a directory.

And not just files, you can also have other sub directories within a directory. And usually, in every computer system there is what is called a directory tree, that is the hierarchy of all the directories and subdirectories and all the files it is arranged like a tree, you can think of it like a tree. For example, it starts with say the root you have slash, in Linux systems there is a slash which is called the root directory.

And underneath the slash you can have home, bin, user, and many different directories under home maybe you have a directory called foo. And in this directory, you have this file a dot txt. So, you can have a path like this slash, this is the root directory slash home slash foo slash a dot txt. So, this is called the root file system. The directory starting, the directory tree starting at the root this is called the root file system.

So, when your system boots up, you will have this root file system setup. And once your system starts to run, you can also attach other directory trees to an existing directory tree. If you have some other files stored on some other disk, you can have slash disk 2, slash some more files, you can attach another directory tree here to some point in the existing directory, that is called mounting a file system.

So, you have files directories, all of which are organized as a directory tree, starting from the root, this is the root filesystem. So, if you remember when we studied containers, we have said that multiple containers work on the same operating system, but they have different root file systems. So, different containers will have a different view of this directory tree, of this root file system.

So, in different containers, you can have different files, different executables, different libraries, and different configuration files, all of this can be different, the view of your file system can be different in different containers. But even though the underlying operating system is the same.

So, this is about the root file system and the file abstraction itself. Now how do you access a file? How do you read and write data into a file?

So, if you have done any basic programming, in any programming language, you must already know this concept. So, to access a file, what do you do? First, you open a file, there is usually a system call in Linux, it is called the open system call. You open a file, you give the file name, and you ask the operating system to open this file for you. Note that this is a privileged operation, this involves accessing the disk.

So, the user program cannot do it, it has to be done via a system call. So, when you open a file, the operating system will return a file descriptor or a file handle to you. What is this file descriptor? This is the handle using which you will refer to this file and say that you want to read or write and so on. So, for example, if you want to read any data from this file, you will say read from this file descriptor, write into this file descriptor.

And when you are done, whatever you want to do with the file, you will close the file. So, this file descriptor is sort of a handle, instead of every time saying you know the full file name or something, you can just say this file descriptor, I want to do this operation on this file. So, this is the concept of a file descriptor. And you have system calls to open, read, write, close files, and so on.

(Refer Slide Time: 07:26)

Reading and writing a file

```
fd = open("/home/foo/a.txt")
char buf[64]
n = read(fd, buf, 64)
buf[0] = ...
n = write(fd, buf, 64)
```

- After opening a file, process can read/write to a file as a stream
 - File descriptor used as handle to refer to the open file stream
 - Read system call reads specified number of bytes into a user-defined buffer, returns number of bytes read
 - Write system call writes specified number of bytes from a user-defined buffer, returns number of bytes written
- Implemented by commands to disk controller via device driver in OS
 - Device driver gives command, data transfer via DMA, device raises interrupt when done
- Read and write system calls update the **offset** into a file
 - After reading N bytes, next read will return the next set of bytes
 - Can also update the offset from which to read/write using a seek system call
- Every open file descriptor will read/write file as independent stream, with independent offsets
 - Exception: parent and child processes share same offset after fork

The diagram shows a vertical stack of rectangular blocks representing a file. A red arrow points from the code snippet to the top of this stack. A red bracket on the right side of the stack is labeled 'offset' with a red arrow pointing to it.

So, let us understand this read and writing of a file in a little bit more detail. So, after you open a file, a process can read and write to the file as if it is a stream of bytes. So, you can think of the file as a stream of bytes and you can read multiple chunks of these, the stream at a time. So, here is just a sample code, again, that you all should be familiar with if you have done any basic programming.

So, you open a file, and then you will read. So, this read system call takes a few arguments. It takes of course, the file descriptor and it takes a buffer, a user buffer, some buffer in your program as an argument. So, here we have defined character array of 64 bytes as our buffer. Buffer is nothing but any memory region into which you can write data. So, the read system call takes the file descriptor, this buffer and how many bytes to read all of these as arguments.

And it returns the number of bytes that have actually been read into this buffer. At the end of the read system call the 64 bytes of data are put into this buffer and then this read system call returns. So, this value of n can be 64 of all 64 bytes that could have been read, but sometimes maybe the operating system could not read all 64 bytes for whatever reason in which case this n could be less than 64.

So, this is the format of the read system call. Similarly, the write system call also the structure is similar. Suppose you have read the 64 bytes of the file into this buffer. And now, you make some changes, the zeroth entry some other entries in this buffer, you change and now you want to write back into the file. Then the write system call takes the file descriptor, this modified buffer that you want to write and how many bytes you want to write as arguments.

And returns the number of bytes that have actually been returned. If for some reason all the bytes could not be returned due to some error, then this n could be less than 64. It could also be something like minus 1 or some error code if the write did not work for whatever reason, we will study all of this when we study how file systems work. So, this is the process of reading and writing a file.

So, if you recollect a lecture on IO devices that we did a few weeks ago, you should now be able to understand what does this read system call do. So, when a process makes a read system call, the device driver, the part of the operating system that talks to the device gives a command to the

device and the data transfer is done by the device via DMA. And once the command is completed the device controller will raise an interrupt to the CPU.

And then the operating system will come and handle the interrupt. So, the device driver tells the device controller hey, read this file data, then the device controller will read the data from disk, transfer it to memory via DMA and the device will raise an interrupt when it is done. So, this process we have already seen before. And the other important thing to note is that the read and write system calls update the offset into the file.

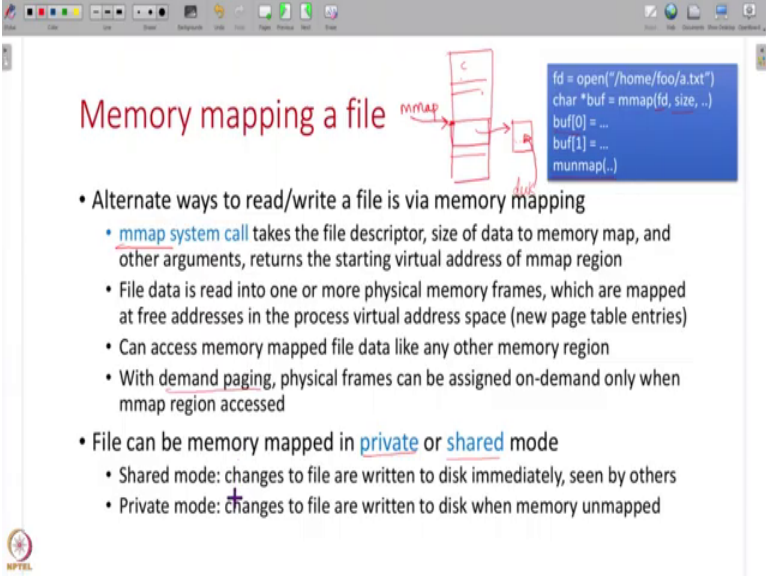
So, if you think of a file as a stream of bytes, after this first read system call of 64 bytes. Now your offset is here. You remember the location in the file up to which you have read. So, after you read 64 bytes, if you say once again read 64 bytes, the next 64 bytes will be returned to you. So, the file remembers okay you have read so far, the offset is remembered and you keep on reading from the current offset.

Of course, you can also update the offset there is a system called the seek system call that actually without reading all of these bytes, you can simply say jump the offset to some other location, that is also possible. And the important thing to note is that every time you open a file, you will have an independent offset. So, if one process opens a file it has read 64 bytes, the offset is here.

If another process also opens the same file and it has read the first 32 bytes, then its offset will be here. So, multiple processes when they read a file they do so with independent offsets, it is like an independent stream of bytes. Just because one process has read 64 bytes when another process opens the same file it will not get the next 64 bytes, it will also read from the beginning. So, the offsets are independent for different processes.

And every open file descriptor basically has its own independent offset. The only exception is when during the fork system call when the parent and child fork then both of them will share the offset that is the only exception. So, that is about reading and writing a file using these read and write system calls. And we are going to study the implementation of these system calls in a little bit more detail next week.

(Refer Slide Time: 12:51)



The slide is titled "Memory mapping a file" in red. It features a diagram showing a file on disk being mapped to a memory region. A blue box contains the following C code snippet:

```
fd = open("/home/foo/a.txt")
char *buf = mmap(fd, size, ...)
buf[0] = ...
buf[1] = ...
munmap(...)
```

- Alternate ways to read/write a file is via memory mapping
 - mmap system call takes the file descriptor, size of data to memory map, and other arguments, returns the starting virtual address of mmap region
 - File data is read into one or more physical memory frames, which are mapped at free addresses in the process virtual address space (new page table entries)
 - Can access memory mapped file data like any other memory region
 - With demand paging, physical frames can be assigned on-demand only when mmap region accessed
- File can be memory mapped in private or shared mode
 - Shared mode: changes to file are written to disk immediately, seen by others
 - Private mode: changes to file are written to disk when memory unmapped

So, the reason why we are discussing the process of reading and writing a file now is that there is actually a different way to read or write a file that involves the mmap system call. And this mmap system call is actually related to memory management, we have seen this a few lectures ago. So, therefore, to introduce this mmap system call as a way to read or write a file we are also now discussing a little bit about files in this week also.

So, the alternate way to read or write a file is read and write system calls are not the only way, there is a different way to do it, which is using the mmap system call. So, here is an example. So, you open a file and then you memory map that file into the address space of the program. What does that mean?

This mmap system call will take this open file descriptor and how many bytes of the file to memory map and a few other arguments and it will return the starting virtual address of this mmap region. That is in the virtual address space of a process, you have the code data stack here various things, and there will also be a free area, some addresses will be free. Into these addresses the contents of this file are memory mapped.

So, the file data is copied into some physical frames. So, you get the file data from disk copied into some physical memory. And the address of this physical memory is stored at these page table entries corresponding to some free virtual addresses. And this starting virtual address is

returned by the mmap system call. So, this mmap system call is returning an address to a pointer to a character buffer or whatever you can cast it into any pointer you want.

It is returning an address to you, at this address you can place all your data or you can read file data, you can write file data starting from these addresses. And these addresses are actually pointing to physical memory frames which have the file data in them. So, it is almost like you can have taken the file data, put it into memory and making it appear like part of the address space of a process.

So, the file data is read into physical memory frames. And these physical memory frames are mapped into the virtual address space of the process. What does that mean? Page table entries are added from these addresses to these physical addresses, in the page table of the process. And now the process can just access this memory mapped area just like it accesses any other data in its memory image.

So, you have a character buffer, you can read and write entries in this character buffer. Everything, you can just manipulate the file just like you are manipulating an array of characters in your program. So, this becomes easy, you are no longer using the read or write system calls. And of course, if you have demand paging, then this physical memory is only allocated when you access like you do an access like this into this memory map region, then only the data is read from disk, it is not read as soon as the user says memory map of file, then it is not read, it is done on demand.

That of course is the concept of demand paging. So, this is a very powerful way to read a file. Now, you have to understand that once the file is memory map, you need not memory map the entire file you can specify what size you want to memory map. And once you have done that, once you have specified that I want the first 4 KB, or first 1000 bytes of my file something to be memory mapped and inserted then the file data just appears like any other variables or data in your program.

So, another important thing to note about memory map, some of the other arguments it takes is the mode in which you can memory map. You can memory map a file either in private mode or shared mode. What does this mean? If the file is memory mapped in shared mode, whenever you

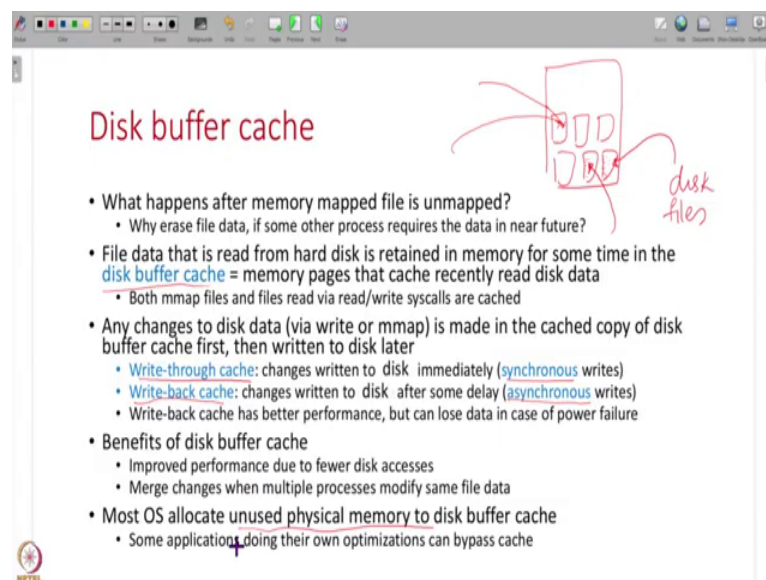
change any of this memory map the region whenever a write is done, this will immediately be done to the disk also.

So, that another process opening this file can also see the changes. But if you memory map it in private mode, whenever you change, the changes are only there in memory. They are not pushed to disk, only when you unmapped, now there is a system called to also unmapped a memory map file. When you unmapped, when you remove this file from memory this way, then only the changes will go to disk.

So, these are two different ways in which your file can be memory mapped. So, now, we have seen that when you memory map a file, the contents of the file are got into main memory. Some physical, into some physical frames, you have copied the file data. Now when the user unmapped the file says, I do not want this memory mapping anymore, I am done, just free up these virtual addresses.

When the user does that, you still have some memory frames, which have the file data in them. So, what do you do with these memory frames? Do you just erase them? It turns out that operating systems do not erase them. So, they try to cache them. Why because the logic is the same, you have gone to disk, disk access takes a long time, you have fetch some file data from disk into main memory, maybe somebody will need the same file again in the near future, why throw it away? You keep it around in memory.

(Refer Slide Time: 18:12)



Disk buffer cache

- What happens after memory mapped file is unmapped?
 - Why erase file data, if some other process requires the data in near future?
- File data that is read from hard disk is retained in memory for some time in the disk buffer cache = memory pages that cache recently read disk data
 - Both mmap files and files read via read/write syscalls are cached
- Any changes to disk data (via write or mmap) is made in the cached copy of disk buffer cache first, then written to disk later
 - Write-through cache: changes written to disk immediately (synchronous writes)
 - Write-back cache: changes written to disk after some delay (asynchronous writes)
 - Write-back cache has better performance, but can lose data in case of power failure
- Benefits of disk buffer cache
 - Improved performance due to fewer disk accesses
 - Merge changes when multiple processes modify same file data
- Most OS allocate unused physical memory to disk buffer cache
 - Some applications doing their own optimizations can bypass cache

So, this is called the disk buffer cache. That is in your memory, you have certain memory frames allocated, which basically store disk data. You have files and other things in disk, their contents are copied into memory. And when these contents are copied into memory, and the process finishes using this disk data, these file data, these contents are retained in memory for some more time as a cache. This is called the disk buffer cache.

And why do we do this? So, that in the future, if some other process reads the same file, we do not have to go to disk again, we have a copy in memory. Anyway, we have gotten it all the way, why not keep it for some more time. And note that this is not just for memory mapped files, even files that you read via read and write system calls, they are also cached in the same way.

So, when you read a file using the read system call the device will DMA the file data into some memory. And that memory, the user process will use. But that memory after the user process finishes reading, this memory is still retained for some more time, in case some other process also reads the same file. So, not just mmap but also read, write system calls.

Any time, any file data is read using any system call it is retained in memory for some more time in the disk buffer cache. So, what about when you write to this data? So, you have memory mapped a file or you have read some file data into memory and then the user will modify this data, will make some changes to this in memory copy of this file, then what do you do? What does this disk buffer cache do?

There are, it can do two things. So, you can have a write through cache, which is if you have memory mapped a file the memory frames, if they are modified, whenever they are modified immediately, you will take it to disk, that is called a write through cache. Or you can have a write back cache, which is you keep the changes in memory for some more time, and the later on you write it to disk.

So, if you have a write through disk cache, then the writes are synchronous writes. As soon as you write into memory, the disk is also updated. If you have a write back cache, the writes are asynchronous writes. That is after you write, after some time, only the disk will change. So, this concept of write through write back caches we have seen with CPU caches also, it is the same concept.

Any time you are caching something, you can immediately make changes to the cache as well as the original copy or you can, make changes to the cache first and update the main master copy later on. That is simply the concept of write through and write back caches. So, which one should operating systems use for the disk buffer cache? So, it turns out that mostly we use write back cache because it has better performance.

As soon as you write you are not running to the disk, because disk writes take a lot of time. So, you can do this a little bit later. So, it has better performance, but you can also lose some data in case of a power failure. Recall that main memory is volatile storage. That is if the power goes off, the user has written to a file, the power has gone off, the users changes could be lost. The user thinks hey, I wrote to a file, to hard disk, to persistent storage.

How come my changes are lost, but that can also happen, in case of some very unfortunate power failures. After the write has been done in memory, but before it can be done on disk. So, that is the trade off with write back and write through caches. But in spite of these difficulties, the disk buffer cache is a very useful thing to have, because usually some common files are read by multiple processes a lot of times.

And, instead of every time going to disk you have a copy in memory is actually very useful. the read system call can actually complete much faster sometimes. So, you improve performance due to fewer disk accesses. And another advantage of disk buffer caches when multiple processes modify the same disk area, all their changes are merged in memory.

It is not like one process right something another process writes something and you have two inconsistent copies of your disk, that will never happen. Because both of them are modifying the same entry in the disk buffer cache. So, this buffer cache will do this coordination of two processes are reading the same file or writing to the same file, it will ensure that one of them writes first, then the other rights the changes are merged, that is ensured by the disk buffer cache.

Because they are both not independently going to memory and writing. Instead, everybody is editing the copy in the disk buffer cache. So, which is why this disk buffer cache is a very important part of modern operating systems. And in fact, modern operating systems any unused physical memory you have, you have 8 GB of RAM about 2 GB are being used for process memory image OS code data and everything.

Whatever memory is free, you will simply give a lot of it to the disk buffer cache. As you are reading files from disk, you will keep storing it in your free memory. Because anyway, the memory is free, I might as well use it as a cache. Of course, some applications do not want to use the disk buffer cache because they themselves will do their own caching and all of that, their own optimization.

So, there is a way to read files without going to the disk buffer cache also. But for most processes, the disk buffer cache is actually very useful. So, when we study file systems, we will come back and revisit this disk buffer cache and understand the role it plays when reading and writing files in more detail. At this point, the only thing I want you all to understand is that the file system also has some footprint in your main memory.

Your main memory is not just for memory images of user processes and OS code and data. But some file data is also stored in physical memory, that is not being used for other purposes in the form of the disk buffer cache. So, this is the important takeaway for this week. And in the next week, we will understand this in more detail, we will revisit the file system.

(Refer Slide Time: 24:37)

Understand various layers of caches

- **CPU caches**: stores recent memory instructions/data accessed by CPU
- **TLB**: stores recent virtual to physical address mappings done by MMU
- **Disk buffer cache**: stores recently accessed filesystem data in memory
- CPU cache and disk buffer cache store actual data, TLB stores mappings
 - TLB hit avoids only page table walk, not actual memory access
- All caches use locality of reference to avoid extra work in future
 - Temporal locality of reference: data accessed in recent past will be likely used again
 - Spatial locality of reference: data around current access will be likely used again
- All caches use some variant of LRU (least recently used) policy for evicting old entries when cache is full

So, at this point you might be, if you are getting confused with there are many layers of caches. I just want to summarize everything out here. So, this concept of caching is actually very widely prevalent in computer systems to improve performance. And in a computer system, in one

machine itself you have many layers of caches. First thing is you have the CPU caches, which store the most recent memory instructions and data accessed by the CPU.

So, you have the CPU and you have the CPU caches. And anytime you go to memory get something, you will store them in CPU caches. Then you have the TLB, which actually caches the virtual address to physical address translations. And then, you have the disk buffer cache, which caches recently read file data, file system data is cached in memory in the disk buffer cache. So, these are all the different types of caches in a computer system.

There are of course, small differences. For example, the CPU cache actually caches memory contents. The disk buffer cache actually caches the actual file contents, whereas the TLB does not cache any memory or disk. It only caches the page table entries, the address translations. So, even if you find your address in the TLB, you still have to go to memory to fetch the actual memory with the physical address.

Whereas if you find your address in cache, you do not have to go to memory. If you find your file and disk buffer cache, you do not have to go to disk. So, there are slightly different types of caches. But all of them work on the same principle of locality of reference. That is, if you are, there is temporal locality of reference. If you have access something in the recent past, you will likely use it again, there is spatial locality of reference.

If you have, accessed some part of file, then the parts before and after it within the same block, you will access it again. In a cache line, if you have access this byte likely you will access the bytes here and here again. So, with respect to space, with respect to time, there is locality of reference. That is whatever you did right now, you likely repeat it in this future. Again, there is a for loop or you are running through the same function.

Again, there is a recursive function, whatever. There are many instances where regular code has locality of reference. Therefore, these caches work on this principle, if you are using the same thing, again, and again, let us keep it around for the future. And all caches, of course, the space will run out, the CPU caches cannot store everything that is there in main memory. Disk buffer cache cannot store all files that are there on disk.

Therefore, when they run out of space, they will use some policy like LRU, or least recently used policy for evicting entries, when the cache is full. So, all caches more or less, the point I am

trying to make is more or less, they work on the same principles, even though they are caching different entities or different pieces of information.

So, now to understand, now that we have seen this disk buffer cache and everything, let us go back to this two ways of accessing a file using mmap versus read write system calls, which one is better?

(Refer Slide Time: 27:52)

mmap vs. read/write syscalls

- mmap can be used for file-backed as well as anonymous pages
 - Physical frame mapped into address space can be empty frame or with file data
- Memory mapping a file is an easy way to read file data
 - Executable code, shared library code are memory mapped into virtual address space
- Memory mapping a file avoids extra data copies
 - Read/write system calls read data first into memory (disk buffer cache), then copy from disk buffer cache into user provided buffer
 - Memory mapping a file copies file data into free physical frames, which are directly accessed by user using virtual addresses
- Memory mapping allows reading disk data in large page-sized chunks
 - Useful when reading/writing large amounts of data from file
 - Not very efficient when reading files in small chunks

The diagram on the right shows a memory buffer (labeled 'buf') being read from a disk. An arrow points from the disk to the buffer, and another arrow points from the buffer to a 'read(buf)' function call. A red '+' sign is next to the buffer, indicating a direct mapping or copy.

So, the mmap system call one thing you have to realize is it can be used for files, as well as for anonymous pages, for regular pages. We have seen this before, if the user wants to store data, like in a heap, or stack or something, you can also get an anonymous page via mmap. And when you mmap a page in your address space, there are some free virtual addresses. At these virtual addresses or physical frame will be mapped.

This address will be added to these page table entries and this starting virtual address will be returned to you. So, the mechanism of mmap is the same, whether you store file data here or if this is an empty frame, to be used to store an anonymous page, whatever it is, the mechanism of mmap is the same. And memory mapping of a file is actually very easy way to read file data is widely done, your code executable, shared library code.

A lot of files on disk, when you read them, you will actually you are actually doing memory mapping without you realizing it. And memory mapping of a file what it does? It avoids extra data copies, so you will read the file data from disk once into some memory. And then, the user

will directly access this memory using these virtual addresses. So, there is only one copy from disk into main memory.

On the other hand, if you use read and write system calls, what is happening, you are copying from disk once into memory. And then, the user is giving another buffer. The user is giving some buffer. And from this OS memory into this buffers for example, if the user only wants to read 10 bytes, it will only give a 10 byte buffer. But from this memory page from this 4 kb page, those 10 Bytes you will copy into the user's buffer.

So, there is an extra data copy when you do read write system calls. But when you memory map a file, there is no extra copy. Whatever memory is there into which the disk data is copied that only is accessed by the process using some virtual addresses. Whereas with read write calls, you first copy into memory into like the disk buffer cache and then call from that into whichever buffer the user has provided.

So, there is extra data copies which are avoided by memory mapping the file. And the other advantage of memory mapping is you can read a file in large page size chunks. When you memory map you can get multiple pages of the file into memory and directly read them. But of course, if you are only reading 5 bytes, 10 bytes, if you are reading a file in small chunks, and there is no point getting an entire page and memory mapping, adding page table entries, all of this is an overhead.

So, if you are reading a file in small chunks read write system calls are better. But if you are reading a file in large chunks, then mmap may be better. So, which is better will really depend on your application.

(Refer Slide Time: 30:40)

Data storage options in real applications

- **Local storage:** store data on local physical machine
 - CPU caches (SRAM) – fast, expensive, small memory close to CPU (~1-10 ns)
 - Main memory (DRAM) – random access memory for volatile storage (~100 ns)
 - Hard disk drive (HDD) – traditional magnetic disk, stores file data in blocks (~ms)
 - Solid State Drive (SSD) – faster option than HDD for files, common today
 - Non-volatile memory (NVM) – persistent memory, faster than hard disk
- **Remote storage:** storage accessible over the network in the cloud
 - Network Attached Storage (NAS) – store data in reliable file storage appliances
 - Databases – relational databases to store relational data durably
 - In-memory key-value stores – stores data in key-value format, distributed over several nodes
 - Distributed file systems – file storage built over a distributed system of nodes
 - Remote memory – DRAM-like memory accessible over the network
- Real computer systems use some combination of local and remote storage to achieve the functional and non-functional (performance, reliability) goals of applications

Handwritten notes and diagram: A pyramid diagram on the right side of the slide is divided into four horizontal sections labeled from top to bottom: 'Cache', 'DRAM', 'HDD/SSD', and 'NVM'. To the right of the pyramid, there is a small diagram showing a box labeled 'NAS' connected to a box labeled 'DB'.

So, now to summarize, we have seen many different ways to store data and applications, so let us summarize all of this. So, on your local machine, there are many different places in which data resides. First thing is you have CPU caches. This is a technology called static RAM or SRAM. It is very fast, but it is expensive. Therefore, you have a small amount of caches of kb or mb of caches close to the CPU.

They can be accessed in a few nanoseconds. You can think of this as a hierarchy, you have CPU caches, which are fast, expensive, small in size, then you have main memory, which is actually called DRAM or dynamic RAM. It is a different technology. This is a random access memory, you can access any byte. But it is volatile storage, it takes longer than CPU caches to access. But it is also cheaper, you have more of it, today, we have few gigabytes of DRAM.

Then you have secondary storage, various types of persistent storage. You have your hard disk, which is the traditional magnetic disk, which has a magnetic disk spinning and a spindle going over it, and so on and the head of the disk moving over it. So, this is your traditional hard disk that takes a few milliseconds to fetch data. But these are cheap, you can have several terabytes of hard disks are very common today.

So, you have hard disks, you also have solid state disks SSDs, which are again very common today. They are faster than the traditional hard disks. And they are also similar, similarly used to store files. Then you also have what is called non volatile memory. This is actually like memory

like DRAM, but a different technology which is persistent. It is not storing data in blocks like a hard disk, but it is closer to main memory, but it is persistent.

So, that is called non-volatile memory. This is a new technology coming up in computers today. So, this is being used like a faster hard disk. Because it is persistent memory, it is being used like a faster hard disk. So, these are all ways to store data persistently in the form of files. Whereas DRAM is for memory images OS code data. And a subset of this is cached in SRAM in CPU caches.

So, this is your various options to store data in your local machines. And now today, a lot of applications run on the cloud on a distributed system. In a distributed system, you have many more options to store data of a program. So, these options of remote storage, we will study in the later part of the course when we study distributed systems, but I just want to introduce the options out there.

So, you have what is called Network Attached Storage today, which are also called NAS boxes. That is you have your computer, you have another NAS box somewhere else, instead of storing files on your disk, you store it in this NAS box over the network, you access it and store it. This box provides you very reliable fast storage and multiple computers can store their files on the NAS box.

You also have databases, you which you can access over the network and store relational data as tables. You have in memory key value stores. That is data has been stored in the memory of other computers in a distributed system. And it is distributed over multiple nodes, but it is stored in memory. So, it will be faster than databases in a key value format. So, you have these options. You have distributed file systems.

That is you are storing data as files, but these are distributed over many nodes so that you have more capacity and better performance. Then you have remote memory, which is actually DRAM like memory on some other node that you can access over the network. It is not in your local machine, but it is almost like DRAM. So, all of these are the different options.

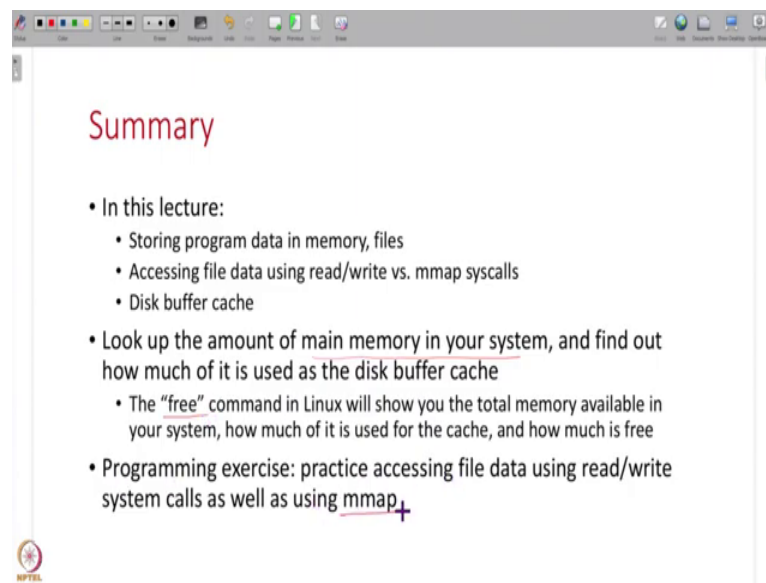
Once you move, look beyond your single machine to a distributed system in a cloud where your application is running. You have many more ways to store data. So, this remote storage part of it we will revisit later in the course in more detail of course, but I am just putting it out here so that

you can understand all of these options. So, whenever you are an Application Designer, you are building an e commerce website or a ticket reservation computer system, any system if you are building you have a lot of data to store, user data to store.

And this to store this data you have many different options should I store it locally, should I store it in a remote storage, each of them has their own tradeoffs with respect to performance reliability, all of these you have to keep in mind to make the decision of where to store data in a user program.

So, that is all I have for today's lecture. In this lecture, we have studied a little bit about file systems, primarily the intersection of file systems and main memory. Because main memory was the focus of our discussions during this week.

(Refer Slide Time: 35:34)



So, we have seen how you can store program data in memory as well as in files and what the intersection of the file system and main memory is in the, in terms of mmap system calls and disk buffer cache and so on. And we have also seen various options for local and remote storage. With the remote storage part being revisited later in the course. So, a small exercise for you is you can understand how much of main memory is there in your system.

There are many commands in Linux for example, you can use the free command that tells you how much memory you have, how much of it is in use by processes, how much is actually being used by the disk buffer cache, how much is free all of this information you can get, please look at

it just to get a feel for things you know, how the disk buffer cache is being used, how much memory is occupying and so on.

And a small programming exercise is also try to read and write files using the mmap system call. The read and write system calls you might have used earlier, but just get more practice using the mmap system call also because this is a very powerful way to access files. So, that is all I have for this lecture. See you all in the next lecture. Thanks.