

Design and Engineering of Computer Systems
Professor. Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 20
Optimizing Memory Access

Hello everyone welcome to the fifteenth lecture in the course design and engineering of computer systems. So, throughout this week we have understood about how programs access memory in this lecture we are going to put all of these concepts together recap them and try to think about how to optimize the performance of the program with respect to memory access. So, let us get started so this is a recap of what we have studied so far spread over multiple lectures in this course, which is when you do a memory access.

(Refer Slide Time: 00:52)

Memory access: CPU caches

- CPU fetches instructions/data from memory of process
 - Faster memory access implies faster application performance
- First step in a memory access: check **CPU caches** if data is present
 - CPU caches store recently accessed memory in 64 byte cache lines
 - Uses **locality of reference** to avoid expensive main memory access
- Multiple levels of cache, some private, some common across cores
 - Memory location is cached in the private cache of one core C0, another core C1 also wishes to access the same memory contents → cache line is shared across cores via cache coherence mechanism
 - Cache coherence protocol ensures consistent view of memory across cores
 - But cache coherence mechanisms add overhead to memory access

Handwritten notes and diagram:
CPU → L1 (64 byte)
L2
L3
main memory
C0 C1
L1 L1 +
L2 L2
LLC L3

When the CPU requests some instruction or data at a certain virtual address what happens, the first step that happens is we will check the CPU caches. Now CPU caches store data either using the virtual address or the physical address but that complication let us ignore for now. So, let us assume that the first step in a memory access is checking the CPU cache if the data is present or not.

And if the data is present in the CPU cache so the CPU checks one or more layers of its caches L1, L2, L3 and so on it will check L1 if the data is not there L2, L3 and so on. If the data is present in any of these caches, then it is a cache hit the CPU has its instruction or data it can

proceed. If not, then the CPU will have to go to main memory to fetch the data, so this is what we have seen.

And the other thing that we have seen is that recently accessed memory is stored in 64 byte cache lines in all of these caches. So, this is in the granularity of 64 bytes and the main principle that the CPU cache uses is locality of reference. If you have recently accessed something, then there is a very likely chance that you want to access it again in the future immediately. Therefore, it is worth keeping it around in these caches.

And if you have a cache hit, then your application performance is going to improve significantly, because memory access takes a long time whereas caches can be accessed much more quickly. So, this is the main idea of CPU caches and there are multiple levels of caches and in these multiple levels some caches are private to a core whereas, some other caches are shared across multiple cores.

For example, typically in today's system you have L1 and L2 cache that are separate for each core say core C0, C1 has its own L1, L2 cache and both these cores share a common last level cache or L3 cache. So this is typically the architecture today different systems can have different architectures. So, when you have these multiple levels of cache some being private and some common across cores you also have a slight complication we have also seen this before.

Which is if C0 has read some memory contents into its L1 cache there is some memory location x here and C1 also wants to access x, it cannot go to DRAM and get it it must get it from C0 because C0 may have a modified copy of this memory location. So, you need some coordination across CPU cores to exchange data and this is called the cache coherence protocol or cache coherence mechanism.

This is needed to ensure a consistent view of memory so if C1 goes to memory and gets this location x again, then it may see an incorrect or an inconsistent or a stale value of the memory location. Therefore, the cores need to be talking to each other C1 needs to know who all have this memory location in their private caches all of this needs to be kept track of which is a slight overhead.

But, modern computers do it in order to provide correct access to main memory. And this cache coherence mechanism of course adds an extra overhead to memory access it is not just checking

the caches if it's a cache hit sometimes checking the cache also requires talking to other CPU cores talking to other caches also. So this is a recap of what happens on a memory access with respect to CPU caches.

Now let us see if you are a programmer you are building a large computer system and you want to optimize the usage of cache in your system how would you go about doing that. Note that cache hit rate is one of the main determinants of performance. If you have a good cache hit rate your application can execute CPU instructions faster. If you have a poor cache hit rate, then your applications performance will be significantly slowed down. So therefore, this is important and how do we do this here are a few tips for you.

(Refer Slide Time: 04:54)

Optimizing cache usage (1)

- Programmer can optimize code to maximize cache hit rates
- Align data structures to cache lines using language library primitives or compiler hints
- Store frequently accessed variables together in the same 64 byte cache line
- Write code such that working set size (frequently accessed code sections or data structures) fit in CPU caches
- Write code to increase locality of reference (access data that is already in cache as far as possible)
 - Example: access matrix along rows rather than along columns
 - Example: merge two for-loops that loop over same array

Handwritten diagrams include: three horizontal bars representing cache lines, a grid representing a matrix with arrows indicating row-major traversal, and a code snippet showing two nested loops merged into one: `for i ...
a[i] + a[i]`.

Of course, this is a detailed discussion by itself but here are some simple things you can keep in mind to optimize the cache hit rates. So, one thing you can do is align data structures to cache lines and there are either programming language primitives or compiler hands many ways to do this. But the main idea is your if you have a data structure try to align it to the start of the cache line.

So, suppose you have you know 64 bytes and the next 64 bytes here do not let your data structure go from here to here. So, because why because when you access this data structure this cache line as well as this cache line both have to be gotten into memory. Instead, write your data structure at an address that is aligned with 64 bytes so that your data structure fits in one cache

line and you can just bring one cache line into one of the caches and your entire data structure is there.

So, how do you pick the address of your data structure to be a multiple of 64 bytes so that it is aligned. There are ways to do it depends on the programming language you are using or the compiler you are using there are different ways to do it. So, the other thing is store the frequently accessed variables together in the same 64 byte cache line.

Suppose there are two integers that you will always access together, then put them on the same cache line so that when you access one the other is also already in cache the next time you access the second one you do not have to go to memory again it will be a cache hit so this is another thing to do.

Then the other thing what you can do is write code such that the working set size you know which is whichever instructions or data structures you are frequently using this working set size try to make sure if possible that it fits into one of your CPU caches. Of course this is not always possible what do you do if you have a large working set size you cannot compromise the correctness of your program but where possible try to work with a small amount of data that fits into your CPU caches.

So, that once you start working with that data from the next time onwards it will be there in cache and it will be a cache hit. So, there are ways to write code to increase the locality of reference. For example, if you have a matrix consisting of multiple rows and columns and it is stored in memory like this all this row then this row then this row it is stored in memory like this.

So, instead of accessing a matrix like this column wise instead of accessing entries like this, what you can do is you can access entries like this so that when you start accessing this entry all of these are also there in your cache line, then when you start accessing this entry all of this is there in your cache line. In this way if you access it row wise you will have a greater locality of reference.

Similarly, another way is if you have a for loop for i equal to something do something on some array and again at a later point you have another for loop again you are doing something on this array. What you can do is you can merge both these because once you have executed this for loop this array is in your cache.

Then similarly, the next operation also do it here itself so that you do not have to you know evict this out of cache and bring it back later. So of course all of this is assuming it does not impact the correctness of your program you should not write wrong logic simply to improve your cache hit rates. But where possible these are some of the things to keep in mind when you are accessing a large data structure try to think can I make my code in such a way that the locality of reference improves.

(Refer Slide Time: 08:43)

Optimizing cache usage (2)

- When accessing data from multiple cores, avoid cross-core cache coherence traffic to make cache access faster
- Threads of program running on separate cores should access data in separate cache lines as far as possible
 - True sharing: two threads read same memory address from separate cores
 - False sharing: two threads read separate memory addresses, but both locations are on the same cache line
 - Both cause cache line to bounce across cores
- Avoid shared data and lock contention between threads as far as possible
 - Shared lock variable accessed from multiple cores, cache line bounces across cores
 - Recent research on locks which avoid sharing lock variables across cores *scalable locks*
 - Lock-free data structures; data structure implementations that avoid locks using clever tricks

Handwritten annotations: At the top right, a diagram shows two cores, T0/c0 and T1/c1, with a double-headed arrow between them and 'x' marks, indicating cross-core access. To the right of the first bullet point, a diagram shows two threads, T0 and T1, both pointing to a single memory location 'x', illustrating false sharing. To the right of the second bullet point, a diagram shows two threads, T0 and T1, pointing to different memory locations 'x' and 'y' which are within the same cache line boundary, also illustrating false sharing.

Now the other thing concerns multi-core access when you have multiple cores to the extent possible avoid cross core cache coherence traffic that is, if you have a thread T0 running on core C0 and you have a thread T1 running on core C1 both of them to the extent possible each core will have its own private caches and so on. So, make sure to the extent possible that this thread is using different set of data and this thread is running on a different set of data because if both of them use the same data then there will be some cache coherence traffic.

So, we want to ensure as far as possible again without compromising correctness as far as possible ensure that the threads of a program access their own separate slices of the data. So, there is a concept called true sharing which is both threads T0 and T1 both are accessing the same variable, thread T0 and T1 both want to access the same integer or variable x that is true sharing.

The other thing you have to watch out is what is called false sharing which is on the same cache line thread T0 is accessing this variable x and thread T1 is accessing this variable y. So, you might think as a programmer or they are accessing different variables so it is okay they are not sharing data.

But both these variables are on the same cache line so therefore what happens T0 will access x then this cache line will be in the private cache of core 0 then when T1 needs to access y again the same cache line is pulled here then the cache line is pulled here cache line is pulled here this cache line will bounce across course. Because caches store data at the granularity of cache lines you will not get just a piece of the cache line into cache you will get the entire cache line.

So, different threads of a program to the extent possible make their data to be on separate cache lines this will avoid bouncing of this cache line across course. And of course, as far as possible just avoid any shared data between threads because when the threads are sharing data you will need some kind of locking.

So, when two threads access the same variable you have to lock you have to acquire a lock you have to wait for a lock this involves its own overheads. Then the lock variable itself we have seen how locks are implemented you use a variable and you use an atomic instruction that updates that variable, so that variable itself can bounce across different course.

So which is why to the extent possible try to avoid locking between threads but then you might ask how do I do it I mean locking is needed for correctness. So, there are many techniques there is some recent research on locks that use different lock variables on different cores so that you do not have this cache line bouncing across course when you try to acquire a lock.

There is some research on these are called scalable locks right so these are called scalable locks because you no longer have this cache coherence between cores and your performance scales with the number of course. There is also implementations of data structures which are called lock free data structures that is these data structures like linked list they will insert delete elements for you in the same shared link list but without using locks they use some clever tricks like hardware atomic instructions but not actually locks so that your locking overhead is avoided.

So, there are this is a active area of research and there are many ideas out there on how to reduce lock contention between threads and if your program is seeing a lot of this lock contention you

might want to explore some of these ideas in order to decrease the amount of cache coherence traffic and decrease the overhead in your system. Now moving on so if there is a cache miss what happens next, so this is the story of memory access that we have studied so far you first check caches if there is a cache miss CPU has to fetch the contents from main memory.

(Refer Slide Time: 12:43)

Memory access: MMU, TLB, page fault

- If cache miss, CPU must fetch instructions/data from main memory
 - MMU checks TLB for virtual to physical address mapping
 - If TLB miss, MMU walks page table to translate address
 - Main memory is accessed using computed physical address
- TLB miss leads to extra memory accesses due to **MMU page table walk**
 - Optimizing TLB hit rate crucial, especially with multi-level page tables
- How to improve **TLB hit rate**?
 - Limit working set size in memory, use few memory pages at any point of time
 - **Huge pages**: can use larger page size in order to have fewer page table mappings
- If OS has not allocated memory to a page, MMU traps to OS for **page fault**
 - Servicing page faults may require multiple disk accesses to swap space
 - Too many page faults: **thrashing**, too much time wasted in swapping to/from disk
 - Avoid thrashing by limiting working set size, clearing up unnecessary memory

Handwritten notes: 4KB, Zombie, MMU, TLB, PA, Mem, and a diagram of a multi-level page table walk.

So when you have to fetch something from main memory first you go to the MMU to translate your virtual address to a physical address MMU will check the TLB. If the TLB is a hit if the TLB check is a hit and you have this virtual to physical address mapping. Then you can directly go to main memory using the physical address.

If it is a TLB miss, then you will have to first go to main memory so MMU checks the TLB. If the address is there in TLB you directly have the physical address you go access your main memory. If it is a TLB miss you will first go to main memory access the page table multiple times to translate the address. And then, you have the physical address then you will once again go to main memory to fetch the data.

Now TLB miss that is why it leads to extra memory accesses due to this page table walk and it is especially a problem if you have a multi-level page table if you have a four level page table MMU has to go to main memory four times to read the different parts of the page table before it can get a physical address using which it will go to memory once again. So, to avoid this MMU page table walk you need a good TLB hit rate.

So, the question comes how do you improve the TLB hit rate. Of course it is the same principle if you work with a smaller set of pages at any point of time if you keep your working set size that is the amount of code or data you are looping over small, then you will only have to keep track of fewer address translations and with great likelihood it will be found in your TLB. So, once again write your code in a way that there is some locality of reference and limit your working set size that is one way to improve your TLB hit rate.

The other way is to use what are called huge pages. Now we have seen that the common page size in operating systems today is 4 kilobytes. But, if you use a larger page size of a few megabytes or a gigabyte if you use a big page, if you use a 4 mb page then what happens your program will have fewer pages and you will have fewer page table entries you will have fewer things to store in your TLB.

So, if you increase your page size, then your TLB hit rate will automatically improve. So this is one way if possible use larger pages in your program to have fewer page table mappings. And therefore, greater likelihood of storing all your page table mappings in the TLB. Of course this is not for every program this is only if you have a large amount of data to store in those huge pages. If your program itself is small then there is no point taking a 1 gb page. So again,, this is only where it is applicable for certain applications.

So the other thing that will happen on a memory access is you know you try to access the page table entry but the OS has not allocated physical memory yet for that page. In which case the MMU will trap to the OS because it is not able to translate the virtual address and that will raise a page fault. Now servicing a page fault once again will require multiple disk accesses because the OS has to free up a free physical frame it has to write something to swap read something from swap right we have seen this before that servicing a page fault is actually a very painful process.

And therefore, if you have too many page faults in your system you have too many pages and too few physical frames to store them in. Then if you access this page you get give it a frame, then you access this page you take it away from this frame is taken away from here given to this guy, it is like musical chairs the same small number of physical memory is going around to multiple pages. If that happens, then your system is in a state that is called thrashing there are too many page faults every access leads to a page fault and you are doing a lot of disk accesses.

So, to avoid thrashing once again the techniques are the same limit your working set size. As a program try to ensure that you are using lesser amount of physical memory as far as possible do not spread your data across many pages try to make it as compact as possible. And also clean up unnecessary physical memory usage.

For example, if you have zombie processes you know the parent has forked a child but not yet reaped it then these zombie processes are occupying a lot of physical memory unnecessarily. So, there could be many such inefficiencies in your system where you are wasting memory so cut out on all these unnecessary memory usages so that you have enough physical memory frames to store your working set size.

If you do not do that you will end up with a lot of page faults. So, this is again one more thing to keep in mind when we are talking about how to optimize memory access. So now, we have seen memory access from CPU caches to the TLB MMU page faults and so on. Now here are a few other general tips to improve your memory allocation and access in a program.

(Refer Slide Time: 17:50)

Tips to optimize memory allocation/access

- DRAM allows random access of memory (jump to any address), but sequential access of memory is better for performance
 - CPU prefetcher predicts which memory will be accessed next (estimates stride length of access) and fetches it into cache
- Sequential access of disk data is better for traditional hard disks
 - Spinning magnetic disk has extra delays for random access
- Pre-allocation of memory is better than dynamic allocation via malloc
 - General purpose malloc that does variable sized allocation can be slow
- Custom memory allocators better than general purpose allocator in some cases
 - Slab allocators are better when dynamic memory allocation is in a few fixed sizes
 - Store data in memory-mapped anonymous pages instead of heap
- Avoid copying memory contents unnecessarily
 - Memory mapping a file avoids copying file data from kernel memory to user buffers
- Later in the course: how to measure performance and identify which optimizations are useful and which are not, via profiling code

For example, if you have DRAM so main memory allows random access that is you can access this byte then jump to some other byte then jump to some other while right you can randomly jump across main memory there is no read reason to read it in sequence. But, when possible try to access your main memory in a sequential manner.

Why is this, if this is your main memory and you are accessing it in a sequence then your CPU can actually predict what memory you will access next and then it can pre-fetch it into cache. That is if you are accessing memory with a certain what is called the stride length that is you know you are accessing this byte, then 4 bytes 4 bytes 4 bytes or 8 bytes 8 bytes depending on if you have a fixed stride length.

And you are accessing memory in a periodic manner then your CPU modern CPUs have this thing called pre-fetchers where the CPU will try to predict after this address maybe it will access program will access this address. Then this address, then this address let me predict this and get them into CPU cache beforehand before the actual access happens so that the performance will be better. So, in order to make your CPU pre-fetches work better sequential access is better.

If you are randomly jumping around your program here then here then here then here all over the place and your pre-fetcher cannot predict anything it says i do not know what the process will do next. But if there is a pattern then the pre-fetcher can exploit that pattern to improve your performance. Therefore to the extent possible try to do a sequential access of memory for better performance.

Similarly, the same logic holds for disks also especially for traditional hard disk where there is a magnetic disk that is spinning so once the disk reaches a certain location the next few locations can be accessed very quickly. Otherwise, if your disk has to you know do a random access all over then it will be slower. Therefore, even for disk data the same logic holds to the extent possible store your data sequentially and access it sequentially.

Then another thing to keep in mind is that dynamic memory allocation say via malloc or new anytime you have to dynamically get a small chunk that is slow. Because we have seen malloc there are multiple chunks of memory on your heap malloc has to check whenever you have to allocate say 4 bytes it has to check where is a free chunk that I can give. So, this is in general a slightly slow process dynamic allocation every time you want something instead of going and getting it what you can do is you can pre-allocate memory.

Again where it makes sense in your program just allocate a large chunk say you mmap a large chunk page from the OS, then this is better than every time just doing a small malloc and this can be faster also. The other thing you can do is you can use custom memory allocators once again a

general purpose allocator like malloc that allocates any size you want is useful in small programs but if you are building a large system then this might be very slow. So instead, if you know that your program accesses memory in just fixed sizes or a few different set of fixed sizes then you can use what are called slab allocators we have seen this before.

They organize your heap in a much more optimal fashion in fixed size chunks and it is easier to find a fixed size chunk instead of going over all the variable size chunks. Therefore where it makes sense use slab allocators in your program instead of using the general purpose malloc on the heap.

Then store data in your own memory mapped anonymous pages instead of the heap if your data has a fixed pattern and you know that this is how you are going to allocate it and deallocate it. You might as well get a page from the OS and put your data in it instead of getting small chunks from the heap every now and then.

So, these are all ways in which you can think about when you are doing memory allocation for data structures in your program think about should I just whenever I want just do malloc or can I pre-allocate can I use better memory allocators these are all decisions that you have to make in large systems.

Then the other of course, a very simple common sensical tip to keep in mind is avoid copying memory where possible try to avoid copying memory from one location to the other because this takes time. And one way for example is memory mapping a file if you are reading a large file from disk you copy it once from disk into disk buffer cache or some kernel memory.

And then you simply add a pointer to this page to this physical memory at a certain virtual address space. So the process will access this physical frame which has the file data using these virtual addresses. So this makes it easy you are not copying it once again into any other location. On the other hand if you are using read write system calls it is first fetched into OS memory the device will DMA it and then it is copied into a user buffer that is given as argument to read system call.

So, where possible avoid these extra memory copies and even between two user buffers in your program also you can try to avoid because memory copy takes time. And later on in the course when we study performance engineering how to measure performance how to optimize

performance we will revisit all of this in some more detail. For example, which of these is relevant for your program and which of these optimizations does not make sense for your program. If any optimization just results in a one percent improvement it is probably not worth doing but if some optimization leads to a 30 percent improvement in performance it is worth doing.

So, how much impact do each of these things have on your program you have to profile your code to understand. How to do all of these things, we will study later in the course in more detail. But in this lecture itself since we have just studied memory access I would like to put out all of these ideas in front of you so that you can think about it when you build computer systems.

(Refer Slide Time: 23:45)

p = malloc(20);
free(p);

Common memory-related bugs

- **Memory leak** = memory is allocated via "malloc" or "new", but not explicitly freed by programmer via "free"/"delete"
 - Wastes memory space on heap
- **Dangling pointers** = pointers to memory chunks that have been freed up
 - Pointers to malloc memory after freeing it up, or pointers to stack variables after function returns
 - Accessing such pointers may lead to segmentation fault or incorrect behavior
- Avoid such errors with careful programming, or use language libraries that provide automatic garbage collection via **reference counting** (keeping track of pointers to allocated memory chunks)
 - Example: shared_ptr in C++ is automatically deallocated if all pointers are destroyed
- **Buffer overflow** = overwrite stack content and corrupt stack
 - Allocate `buf[64]` on stack, but read string longer than 64 bytes, overwriting data
- Other errors: misunderstanding pointers, not initializing memory, ..

Diagram: A box labeled 'p' with an arrow pointing to a memory block labeled '20'. Another arrow points from 'p' to a memory block labeled 'buf[64]'.

So, we will end this lecture by discussing a few common memory related bugs. So far, we have seen how to write programs well so that you utilize your memory well and access memory faster. Now here are some of the errors that you can make in your program when accessing memory that you would want to avoid. So, what are some of the common bugs? One of the most common bugs is what is called a memory leak, that is in programming languages in some languages you have to explicitly allocate memory via malloc or new and you have to explicitly free it up using free or delete.

For example, if you malloc some amount say you know 20 bytes malloc and you get a pointer. Then later on once you are done using this memory you have to say free of this pointer. So, when

you do malloc some chunk of 20 bytes is allocated on the heap and that address is stored in this pointer. Later on, you have to tell the heap free up this memory for me otherwise this memory will stay allocated to you. And if you forget to free it up then this memory on the heap will be wasted this will be like a hole on your heap.

So, the other bug is what is called dangling pointers that is you have allocated memory on the heap you have stored it in a pointer and then you have also freed it up and then you try to access that memory again. Now that memory might be given to somebody else or this area of the heap might be freed up anything can happen. If you access such dangling pointers then you might get a segmentation fault or you might be accessing some other memory incorrect behavior all sorts of things can happen.

So, these are called dangling pointers after freeing up now after doing this you access that pointer again. So, these are all issues because of programmer errors that is you are not keeping track of what you have allocated on the heap properly. And but if you want to avoid these errors all together there are actually mechanisms in programming languages today that help you keep track of which memory has been allocated in the heap and when it can be freed up that is called that technique is called reference counting.

Which is if there are pointers to some memory on the heap the heap itself will keep track okay there are pointers here and once these pointers are no longer there to this memory the heap itself will keep track of these references count these references when the count goes to 0 the memory will be automatically garbage collected or deallocated. So, for example, if you use what are called smart pointers like the shared pointer in C plus plus, then the memory will be automatically deallocated once all the pointers are gone to that malloc chunk.

So, if you do not want to have these kind of memory leaks or dangling pointers in your program, you might want to explore using these different kinds of pointers in your programming languages like C plus plus. Then the other common bug is what is called buffer overflow that is you know you have your stack, on your stack suppose you have an array of 64 bytes. And you have various other things on your stack like return address other arguments local variable something.

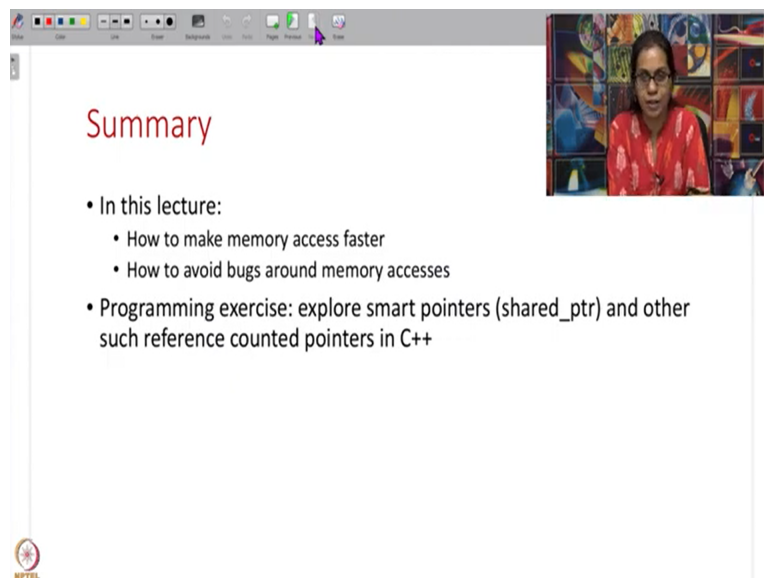
And then, if you read a string longer than 64 bytes into this array, then what happens that string will overwrite all the other elements on the stack as well. So, this is a common issue the

corruption of the stack because the stack has multiple data items stored one below the other and if one of them overwrites then important information like the return address everything can be lost right so this is called the buffer overflow. Attack and a lot of attacks on the internet actually exploit this buffer overflow.

So, this is another common bug that you have to avoid so for example if you are reading a string into a 64 byte buffer you have to check that the string size is 64 bytes you cannot simply just write it to for however long it wants to. So, there are various other errors pertaining to memory like this concept of pointers the fact that the pointer stores an address, what is a pointer, understanding pointers correctly, initializing memory and if you do not do any of these things there are many other errors that can happen with respect to memory access in a program.

And if you have taken a good introductory programming course a lot of these things should have been explained clearly to you. So, that is all I have for today's lecture in this lecture we have basically done a recap of what are all the steps in the memory access when the CPU tries to fetch an instruction or data from a virtual address what are all the steps that happen and how to make each of these steps go faster so that the performance of your application can improve.

(Refer Slide Time: 28:32)



Summary

- In this lecture:
 - How to make memory access faster
 - How to avoid bugs around memory accesses
- Programming exercise: explore smart pointers (`shared_ptr`) and other such reference counted pointers in C++

And we have also seen what are some of the common bugs around memory access and how they can be avoided. So, as a small programming exercise you might want to explore things like smart pointers if you are using languages like C plus plus explore these smart pointers and other such

reference counted mechanisms because they make it easier for you to avoid the common bugs like memory leaks and dangling pointers in your code.

So, thank you all that is all I have for today's lecture we are going to start a brand new topic next week onwards pertaining to the I/O subsystem of files and network and so on in the next week.

Thank you all.