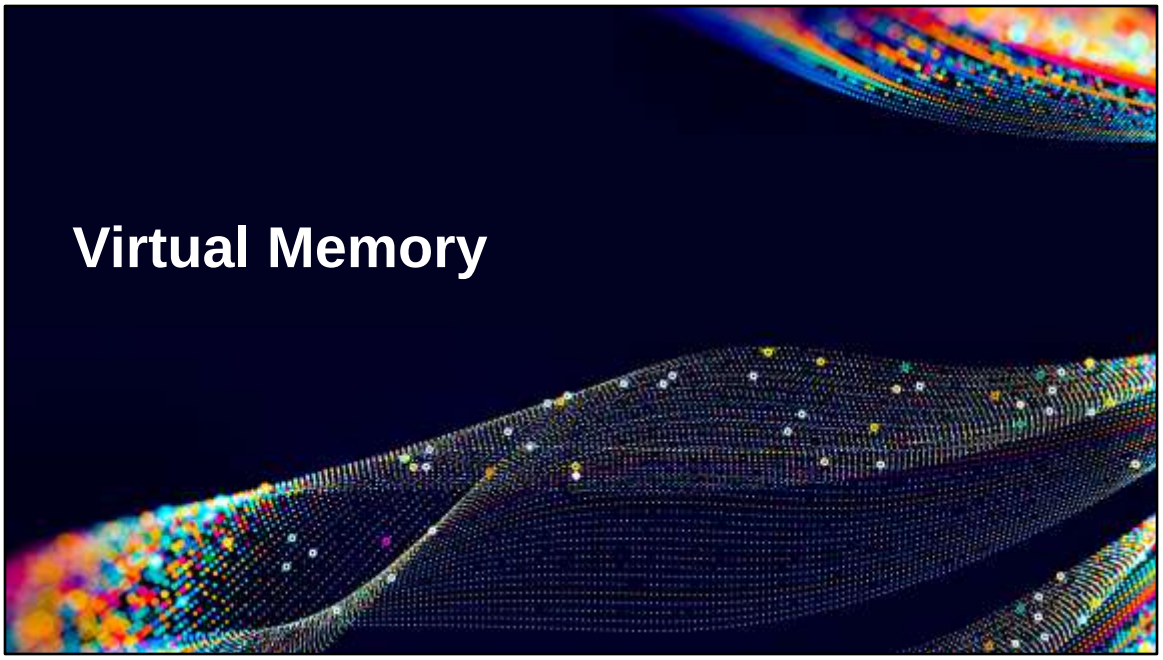
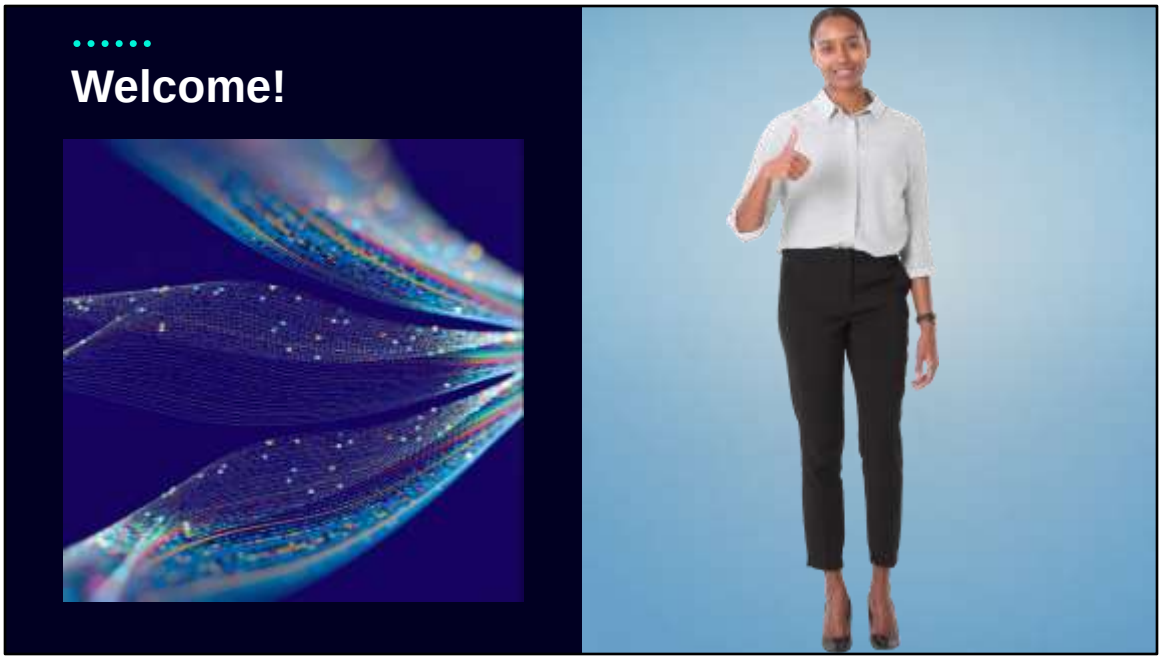
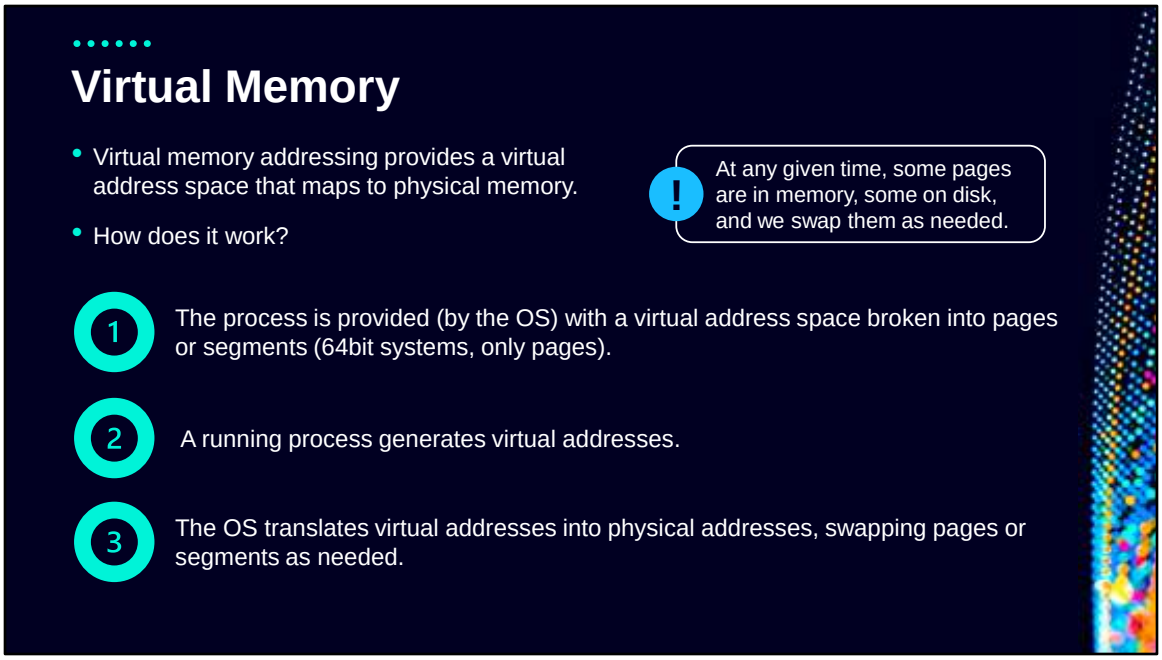

Virtual Memory



ON SCREEN

Welcome back! In this video, we will look into Virtual memory. Let’s get started!

[CLICK – Next Slide]



OFF SCREEN

Now let's talk about virtual memory. **[CLICK]**

When virtual memory is used, processes have access to a virtual address space that is typically much larger than the physical address space existent in the hardware.

The virtual address space is mapped to physical memory and the conversion from virtual to physical is done by hardware. **[CLICK]**

But how does it work? **[CLICK]**

The OS provides processes with a virtual address space broken into pages or segments. (64bit systems typically use only pages.) **[CLICK]**

When a process needs to access memory, it generates a virtual addresses and passes it to the Memory Management Unity or MMU. **[CLICK]**

The MMU translates virtual addresses into physical addresses. If needed pages or

segments are loaded from main memory to disk and vice-versa. **[CLICK]**

Notice that at any given time, some pages are in memory, some are on disk, and we swap them according to what is needed.

[CLICK – Next Slide]

Virtual Memory

Paging

- Page sizes are fixed and are the unit of allocation to processes (typically 4k).
- This may generate internal fragmentation, but no external fragmentation.

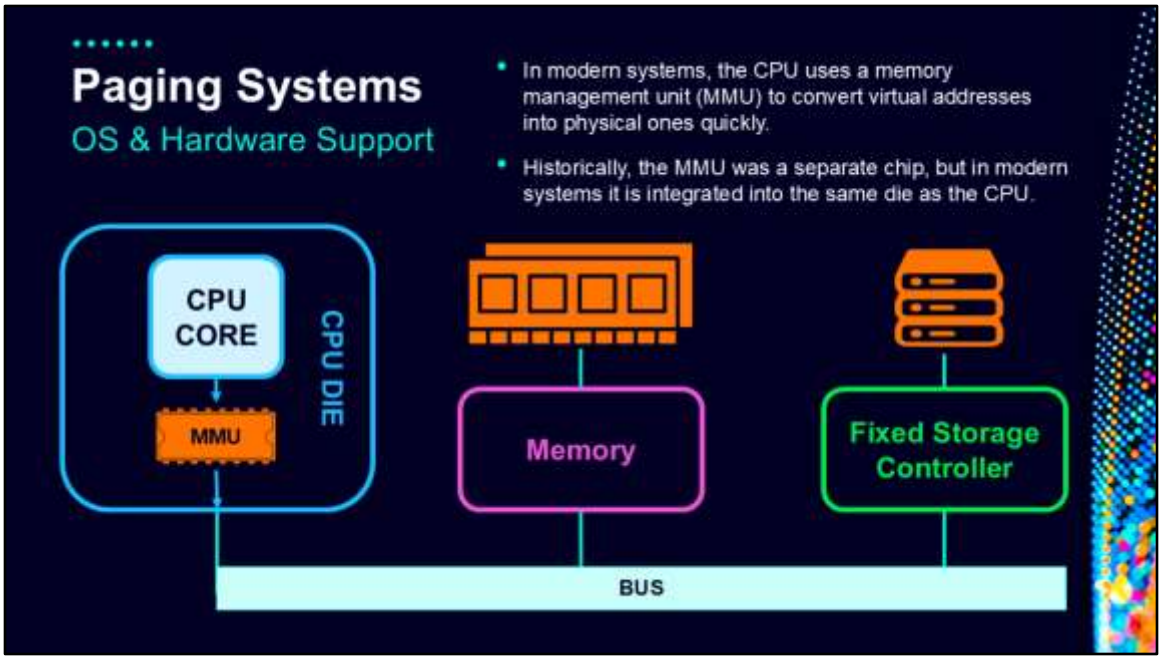
OFF SCREEN

Now let's begin talking specifically about virtual memory using Paging. **[CLICK]**

With paging, the page is the unit of allocation of memory.
The size of a page is fixed and is typically 4k. **[CLICK]**

Note that because pages are fixed in size, external fragmentation is not possible. The memory space is divided into fixed partitions of 4k each, without unusable space between them.
However, we can still have internal fragmentation.
A 4k page may have 3.99k used and the remaining space may not be useful. That is a case of internal fragmentation.

[CLICK – Next Slide]



OFF SCREEN

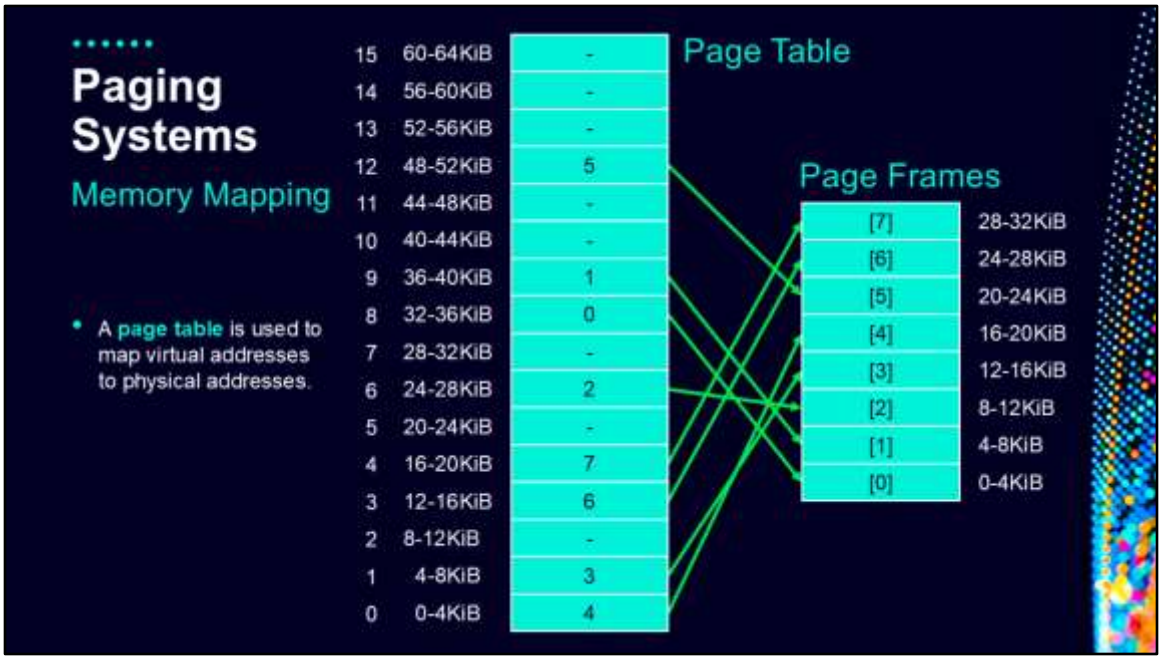
Paging requires OS and hardware support. **[CLICK]**

In modern systems, the CPU uses the memory management unit (MMU) to convert virtual addresses into physical ones quickly. **[CLICK]**

When a process attempts to access memory, it uses a virtual memory address. That virtual address is passed to the MMU. The MMU converts the virtual address into a physical address and puts it on the bus. Physical memory is than read and the data becomes available to the process. **[CLICK]**

Note that historically, the MMU was a separate chip in the main board, but in modern systems it is integrated into the same die as the CPU.

[CLICK – Next Slide]



OFF SCREEN

To map virtual memory into physical memory, a Page Table is used. **[CLICK][CLICK]**

Blocks in physical memory containing pages are called Page frames. **[CLICK][CLICK]**

So, the Page table contains references from virtual pages to page frames in physical memory. **[CLICK]**

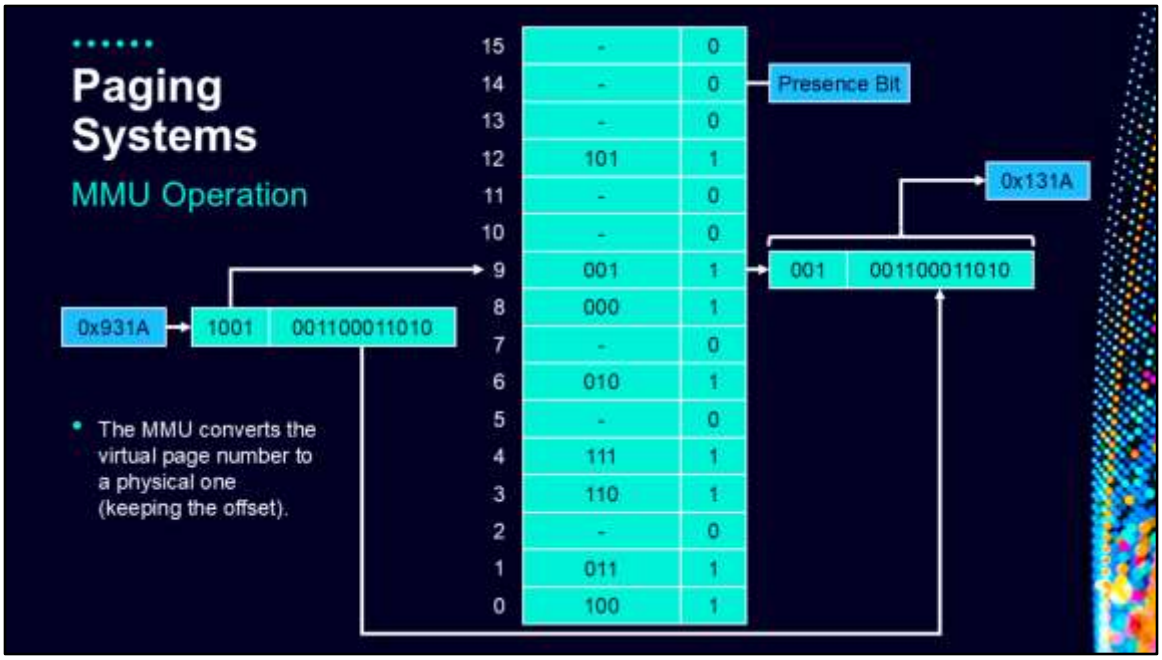
In fact, each line of the page table contains the index of the corresponding page frame in physical memory.

In this simplified representation of the Page Table, the page table can have at most 16 entries and physical memory can have at most 8 page frames.

The fact that the virtual memory space represented by the page table is larger than the physical memory space makes sense.

One of the goals of having virtual memory is to allow processes to access more memory than what really exists physically in the memory hardware.

[CLICK – Next Slide]



OFF SCREEN

Given as virtual address, how do we convert that to a physical address? That is an easy task. We will do it now. **[CLICK]**

But remember,

t happens during every memory access, and is done really fast! **[CLICK]**

For the purpose of address conversion, we use binary numbers. So, we can ignore the hexadecimal representation for now.

Given a virtual memory address in its binary representation, the initial bits from left to right define the index of the page table entry that we are looking for. In this case, 1001.

That page table entry contains the address of the page frame in physical memory

where that page is stored. **[CLICK]**

The rest of the bits in the virtual memory address contain the offset, or the position of the byte we want to access inside the 4k page.

Our task is to replace the index in the virtual memory address by the index of the page frame. **[CLICK]**

But how many bits in the virtual memory address are used to define the virtual page index?

If we look at our page table, it contains 16 pages. To address 16 pages, we need 4 bits. Therefore, the first 4 bits of the virtual address represents the page table index. **[CLICK]**

Note that the offset of the virtual address has 12 bits. **[CLICK]**

That's precisely the number of bits necessary to address the position of each byte in a 4k page.

In the example, the first 4 bits are 1001. **[CLICK]**

That represents 9 in decimal. That is the index in the page table that we must look for. **[CLICK]**

Looking at the contents of entry number 9 in the page table, we find 001. **[CLICK]**

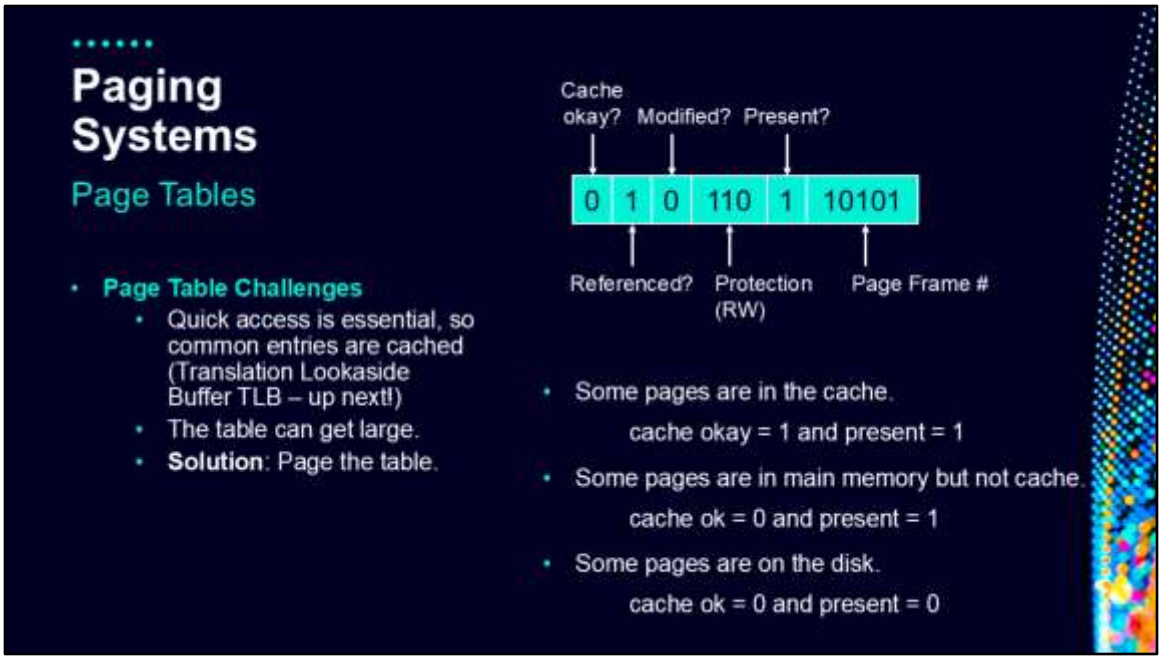
To convert the virtual address to physical address, all we need to do is replace the first four bits from the virtual memory address (1001) by the 3 bits that we found inside line 9 of the page table (001). **[CLICK]**

And copy the offset. **[CLICK] [CLICK]**

Finally, note that there is one bit to the right of the page frame index in the page table. That is the presence bit. **[CLICK]**

When the presence bit is set (or equal to 1), it means that the page frame is in main memory. When it is set to zero it is not, meaning that it must be read from storage.

[CLICK – Next Slide]



OFF SCREEN

So far, we have seen a simplified version of entries in the page table.

Page table entries contain several fields aimed at describing several attributes of the physical pages they point to.

From right to left, page entries have: **[CLICK]**

The page frame number, which is the index in physical memory where the page frame is located. **[CLICK]**

The presence bit, that indicates the page is in main memory. **[CLICK]**

The protection bit, which defines if the protection of the page, if the page can be read or modified. **[CLICK]**

The modified bit, which tells the system is the page frame has been modified **[CLICK]**

The Referenced bit, which tells the system if the page has been read (or referenced) **[CLICK]**

And the cache bit, which tells the system if that page entry has been cached. **[CLICK]**

Note that by looking at the page entry we can see that at any given moment, some pages are in cache (cache ok = 1, presence 1), some pages are in main memory only (cache ok= 0, and presence = 1), and some pages only on disk (cache ok = 0, presence bit = 0). **[CLICK]**

But why do we cache page table entries?

Well, there are several challenges associated with implementing page tables and paging in general.

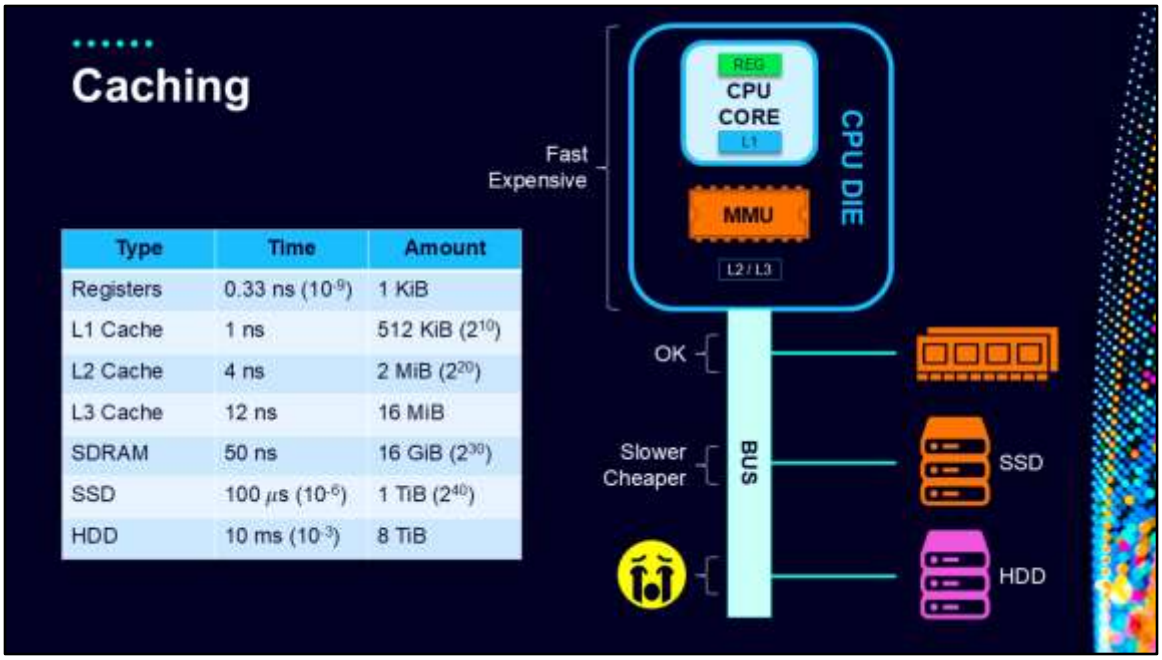
One of them is that quick access is essential! So, frequently used page table entries are cached. **[CLICK]**

In fact, the caching of page table entries is so important that they have their own special cache. It is called Translation Lookaside Buffer. **[CLICK]**

Other problems associated with Page Tables is that they can become really large.

There are different approaches to this problem, but one of them is to page the page table itself. We will look into these approaches later.

[CLICK – Next Slide]



Type	Time	Amount
Registers	0.33 ns (10^{-9})	1 KiB
L1 Cache	1 ns	512 KiB (2^{10})
L2 Cache	4 ns	2 MiB (2^{20})
L3 Cache	12 ns	16 MiB
SDRAM	50 ns	16 GiB (2^{30})
SSD	100 μs (10^{-6})	1 TiB (2^{40})
HDD	10 ms (10^{-3})	8 TiB

OFF SCREEN

Let's talk a little bit about caches.

Remember that cache is a type of memory that is usually small, expensive, but extremely fast.

In the memory hierarchy, we see that the farther memory is from the CPU, memory is larger, cheaper, but also slower.

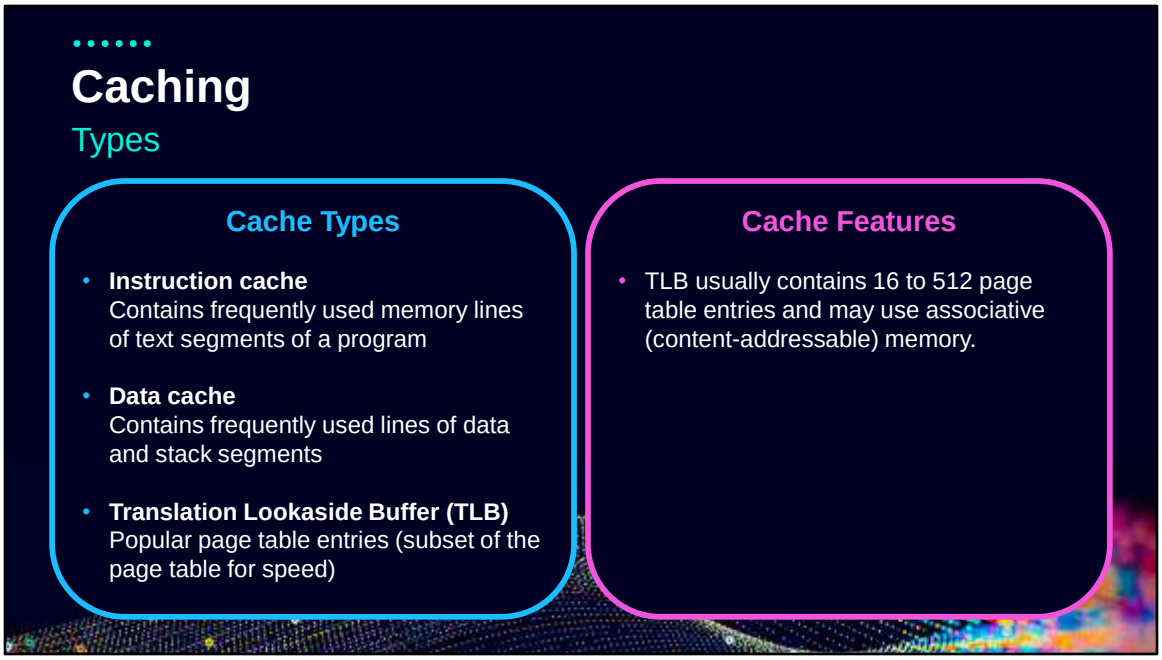
SSDs are extremely slow if compared to Registers.

Examples of cache are the L1, L2, and L3 caches typically present in modern CPUs.

Note that they are really fast, with access times around 1, 4, and 12 ns respectively.

The only type of memory that is faster than the L caches are registers, which are accessed directly by the CPU.

[CLICK – Next Slide]



OFF SCREEN

There are also different types of caches. **[CLICK]**

The Instruction cache is used to cache frequently used lines of text segments of a program. **[CLICK]**

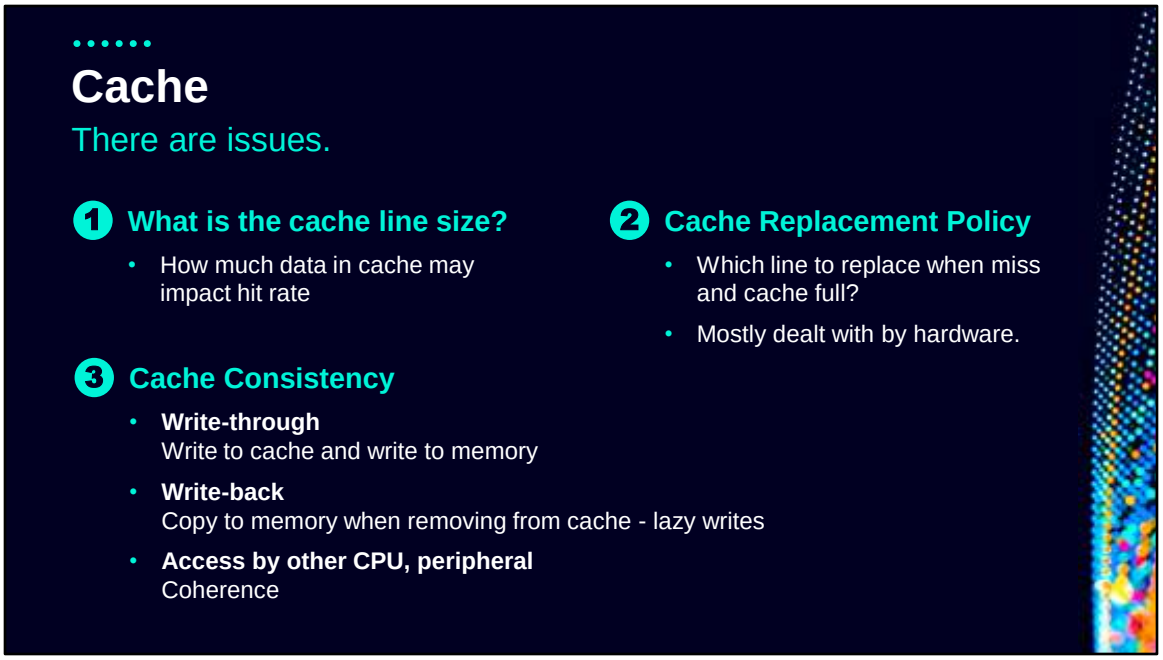
The Data cache is used to cache frequently used lines of data and/or stack segments. **[CLICK]**

More important to us, the Translation Lookaside Buffer or TLB is used to cache popular page table entries. It contains a subset of the page table to speedup access and conversion of virtual pages into physical pages. **[CLICK]**

The TLB usually contains 16 to 512 page table entries and may use associative (content-addressable) memory.

With associative cache, stored data can be identified or accessed by the content and the time required to find an object stored in memory can be significantly reduced.

[CLICK – Next Slide]



OFF SCREEN

While cache can save access time, since they are faster than main memory, there are three main issues with caching. **[CLICK]**

One is to define the cache size. We mentioned that the TLB holds between 16 and 512 lines. Finding the optimum number of lines that will yield a high hit rate while being as small (and cheap) as possible is a challenge. **[CLICK]**

Another challenge is the cache replacement policy. In other words, which line to replace when there is a miss (line not in cache) or the cache is full. There are several algorithms and approaches to these problems. However, there isn't a single answer. **[CLICK]**

The third issue is cache consistency. Cache consistency is concerned with how we make sure that the information in cache is synched with the information in main memory. There are two main approaches: **[CLICK]**

With Write-through every time we write to cache we also write to memory. This can

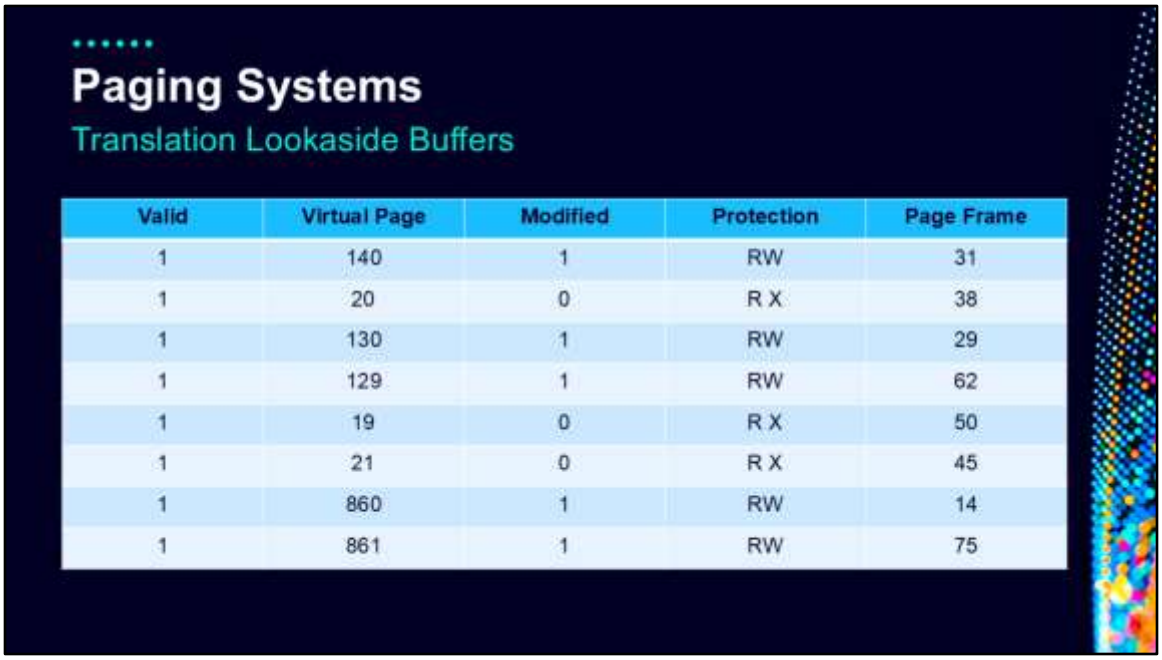
be computationally expensive since writing is dependent on main memory speeds. However, it assures that cache and memory are synchronized at all times. **[CLICK]**

With Write-back we copy to memory when we remove a line from the cache. This is also called lazy writes. This is a faster approach, but it assumes that cache and memory may contain different information while the data is on cache. **[CLICK]**

All these issues must be dealt with for the TLB cache.

Let's keep investigating the TLB in more detail

[CLICK – Next Slide]

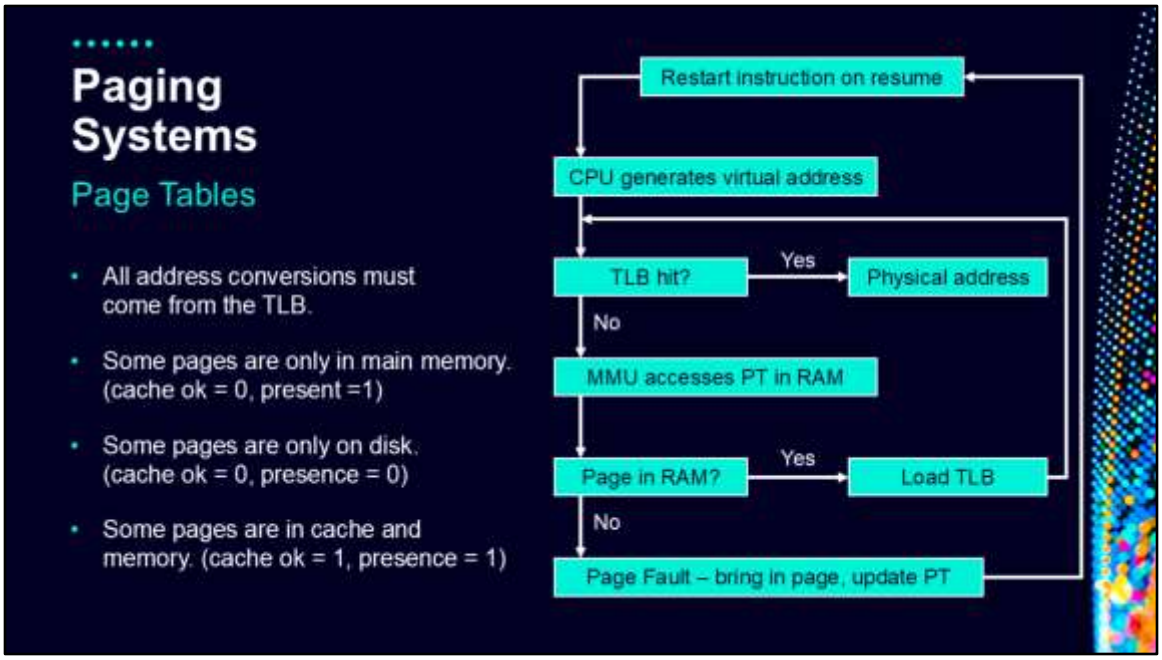


OFF SCREEN

Now let's look at a sample TLB line.

Note that it is very similar to a page entry. This is because the TLB is a special cache, designed specifically to hold page table entries.

[CLICK – Next Slide]



OFF SCREEN

Now, let's look at how pages are accessed and virtual addresses are converted to physical addresses from an OS and process model points of view.

One key point is that all address conversions must come from the TLB.

This is an interesting restriction.

Also, we must keep in mind that some pages are only in main memory, some pages are only on disk, some pages are in cache and memory.

Let's look at the example **[CLICK]**

We start when a process running in the CPU attempts to access a virtual address. **[CLICK]**

So, the CPU generates that virtual address.

If the virtual address is in the TLB cache, the physical address is returned, and the conversion is complete. **[CLICK]**

If the virtual address in not in the TLB, the MMU must access the Page Table in the main memory. **[CLICK]**

If the page is in RAM, **[CLICK]**

we first load the page entry in the TLB **[CLICK]**

and access the TLB to obtain the physical address. **[CLICK] [CLICK]**

But what if the page is not in RAM? **[CLICK]**

Well, in that case, a page must be loaded to main memory from the disk.

That is an IO operation, and therefore, must be caried out by the OS via a system call.

When a page is not on RAM, we say that there is a page fault. **[CLICK]**

The process attempting to access that page is blocked and the page is loaded into main memory via system call. The system call also updates the page table with the entry that is now in main memory.

The blocked process eventually runs again. **[CLICK] [CLICK]**

When it resumes, something interesting happens, it attempts to execute the same instruction, which is to read a virtual memory address.

This is not common. Typically, when there is a context switch and a new process runs, it typically does not repeat the previous instruction. It moves forward. But accessing pages is a special case.

So, upon resuming, the process have the CPU try to access the same virtual memory address. **[CLICK]**

That address is not in the TLB initially. **[CLICK] [CLICK]**

It is in RAM, so it is first loaded into the TLB, **[CLICK]**

and finally, it is read from the TLB **[CLICK]**

and the Physical address is obtained. **[CLICK]**

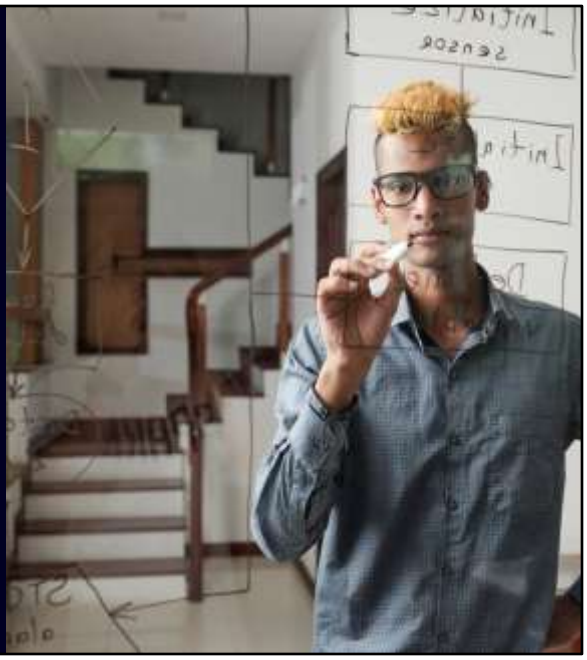
As you can see this is a long process. It may even involve reading from disk, which is always expensive. So, it is important that caches have the right size and replacement policy to minimize cache misses and page faults.

[CLICK – Next Slide]

Really Big Page Tables

How big?

- **64-bit address space**: 16 EiB of Virtual memory space
- With 4KB pages, how many entries in the Page Table?
- $2^{64}/2^{12} = 2^{64-12} = 2^{52}$ page entries (4TB just to store page entries) which is a whole lot of them!



OFF SCREEN

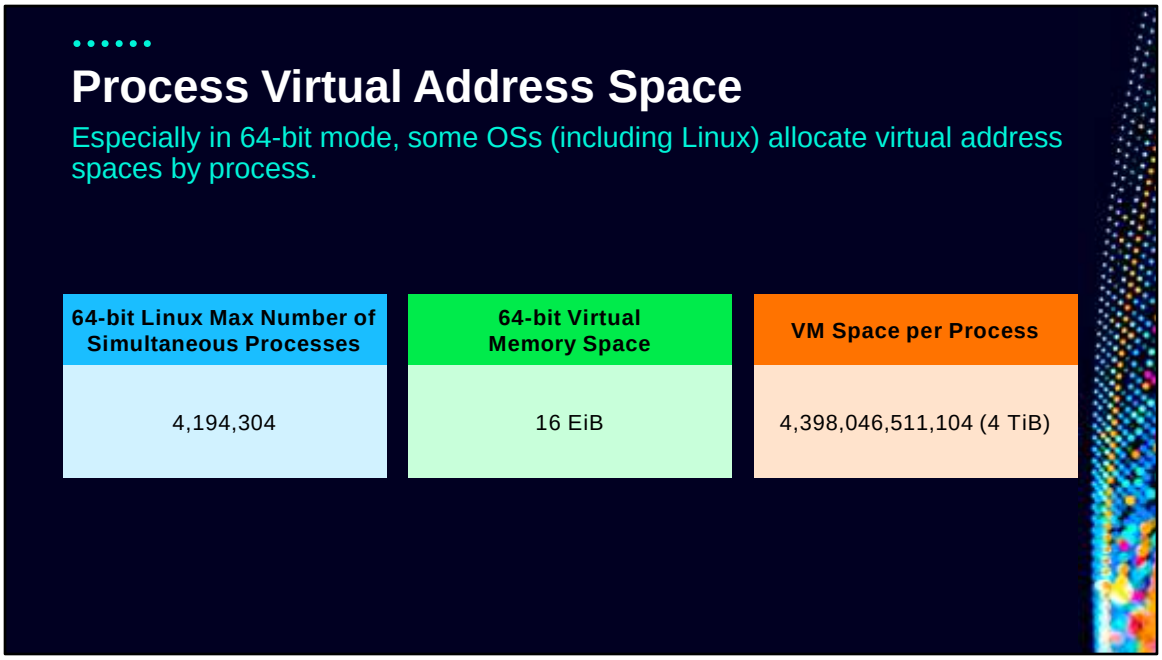
So, now that we know how page tables are used to store page entries and access physical addresses, let's think about how big they can be. **[CLICK]**

With a 64-bit address space, we can address 16 Exabytes of Virtual memory space. This is a huge amount of memory, a lot more than is common to have as physical memory, which is great!! **[CLICK]**

But let's do the math. With 4 KB pages, how many entries can we have in the Page Table? **[CLICK]**

Doing the math, we see that we need 2 to the power of 52 page entries in the page table to store all that. Or 4TB just to store page entries
This is not good! We can't fit the page table in most physical memories!
Imagine that every computer would need 4TB just for page entries, nothing else.
This would also not be smart, since most page entries would never be used.

[CLICK – Next Slide]



OFF SCREEN

So here is a solution:

Some OSs, including Linux, allocate virtual address spaces by process.
The total addressable virtual memory space is divided by the maximum number of processes and page tables are created per process.
This way, we have multiple page tables per process that are smaller and only created when necessary. **[CLICK]**

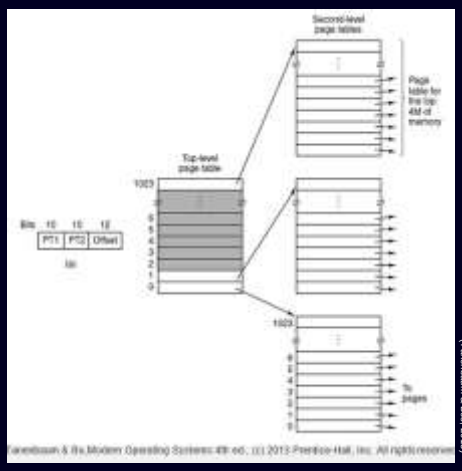
For example, for a 64-bit Linux, the maximum number of simultaneous Processes is about four million. **[CLICK]**

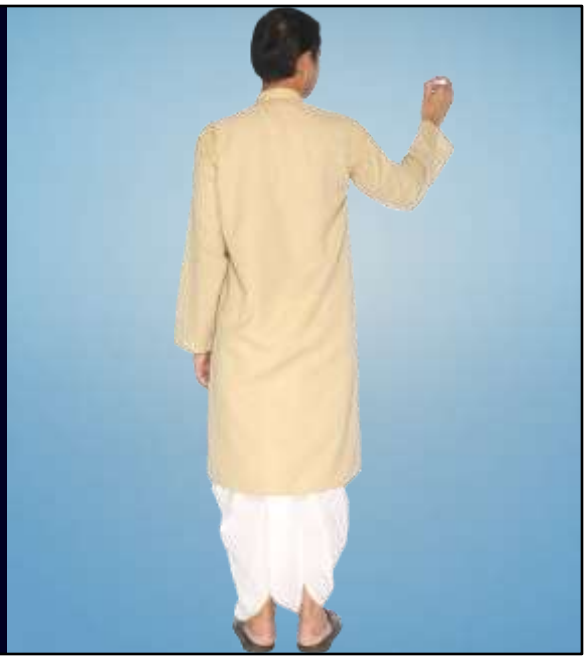
Dividing the total addressable memory space of 16 hexabytes, we now have that each process can access up to 4 Terabytes of memory. **[CLICK]**

This is still a lot of memory for processes, and tables only need to hold entries to address 4TB instead of 16 hexabytes.

[CLICK – Next Slide]

Multilevel Page Tables





ON SCREEN

Still, strategies have been created so that sections of page tables are only allocated when needed.
One of such strategies is called Multilevel page tables.

Basically, with this strategy pages are organized in levels. The example shows a two-level arrangement.

The contents of the pages in level one, point to another page entry, which is a level two page.
The contents of pages in level two, point to actual physical addresses.

With this strategy, pages can be allocated as needed and the indirection of having to access multiple pages is not very costly.

[CLICK – Next Slide]


Thank You!

References

- Biersack, A. (2022). *Computer memory icon* [Online Image]. The Noun Project. https://thenounproject.com/icon/memory-3496697/
- Coquet, A. (2022). *Computer storage icon* [Online Image]. The Noun Project. https://thenounproject.com/icon/storage-2120944/
- ilCactusBlu. (2022). *Computer chip icon* [Online Image]. The Noun Project. https://thenounproject.com/icon/computer-chip-2066136/
- Tanenbaum, A. S. & Bos, H. (2015). *Modern Operating Systems* (4th ed.). Pearson.