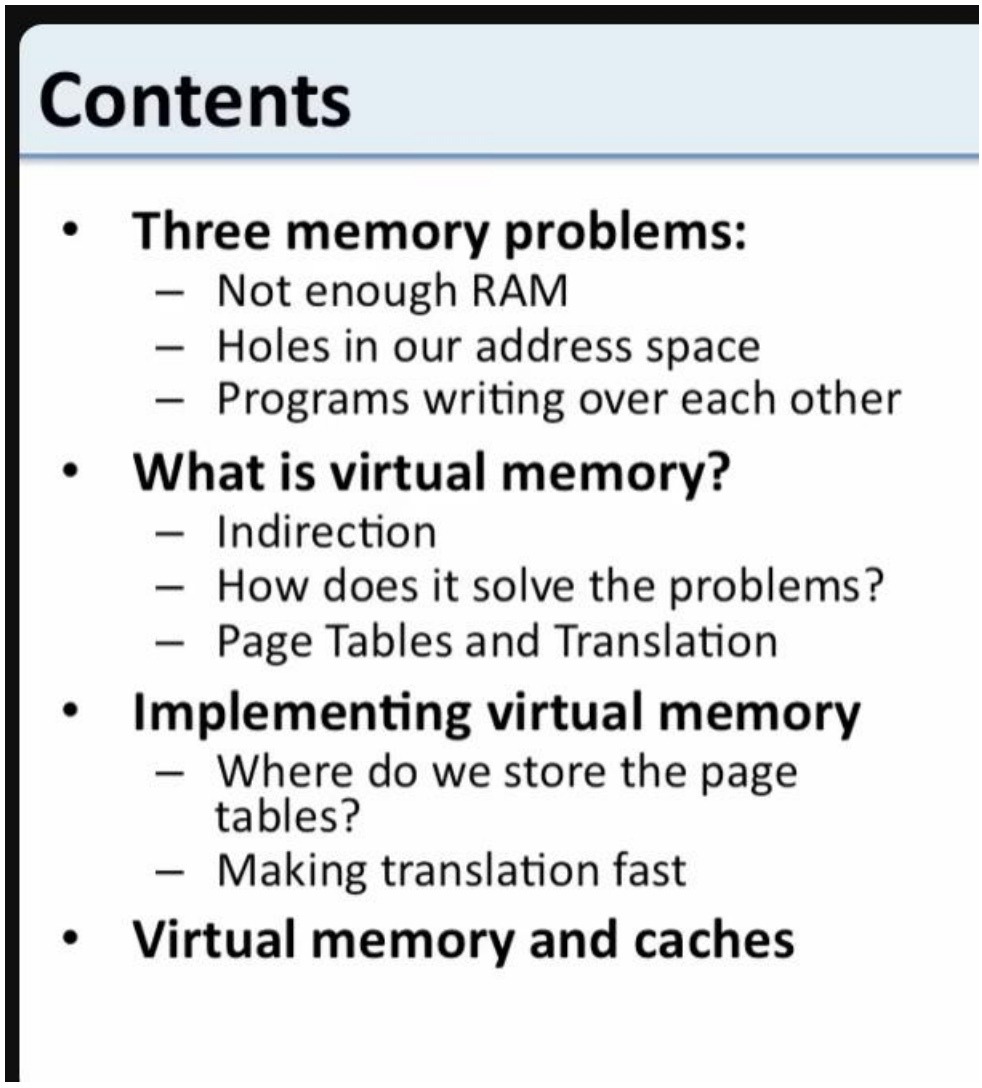
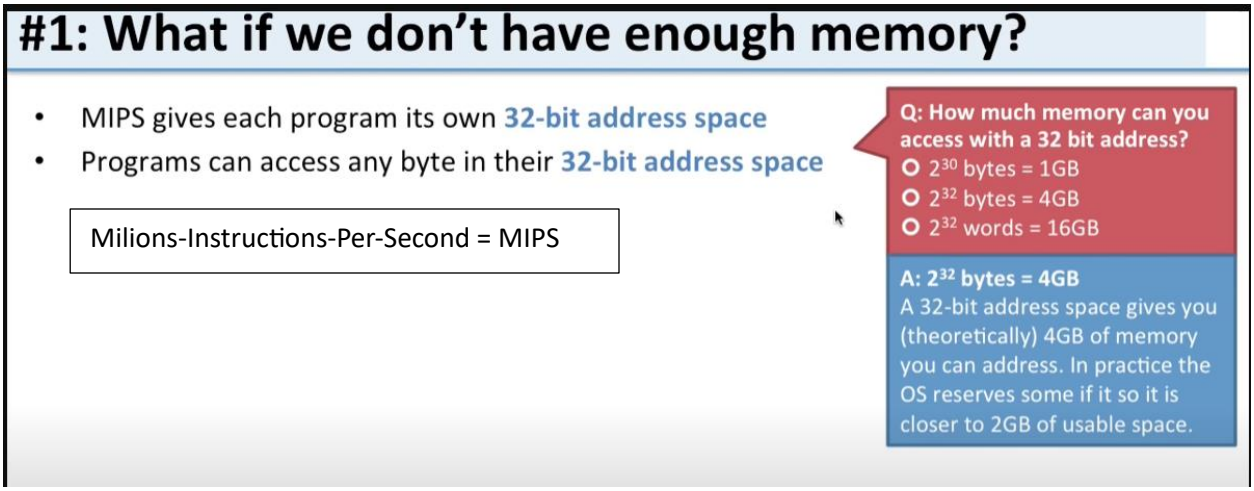
Virtual Memory:1 Introduction

https://www.youtube.com/watch?v=qcBIvnQt0Bw&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=1



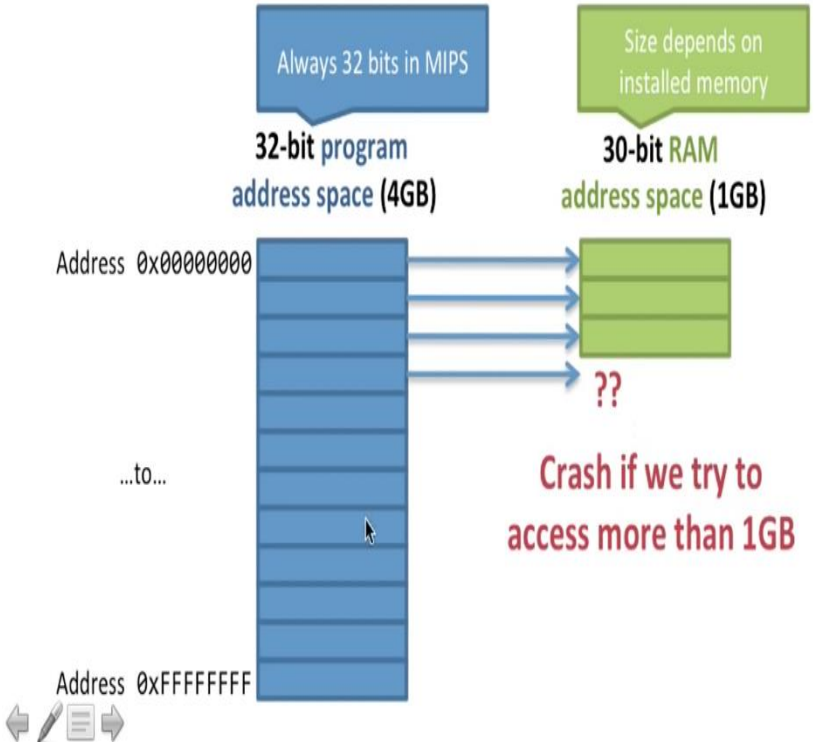
Virtual Memory: 2 Three problems with Memory

https://www.youtube.com/watch?v=eSPFB-xF5iM&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=2



#1: What if we don't have enough memory?

- MIPS gives each program its own **32-bit address space**
- Programs can access any byte in their **32-bit address space**
- What if you don't have 4GB (2^{32} bytes) of memory?

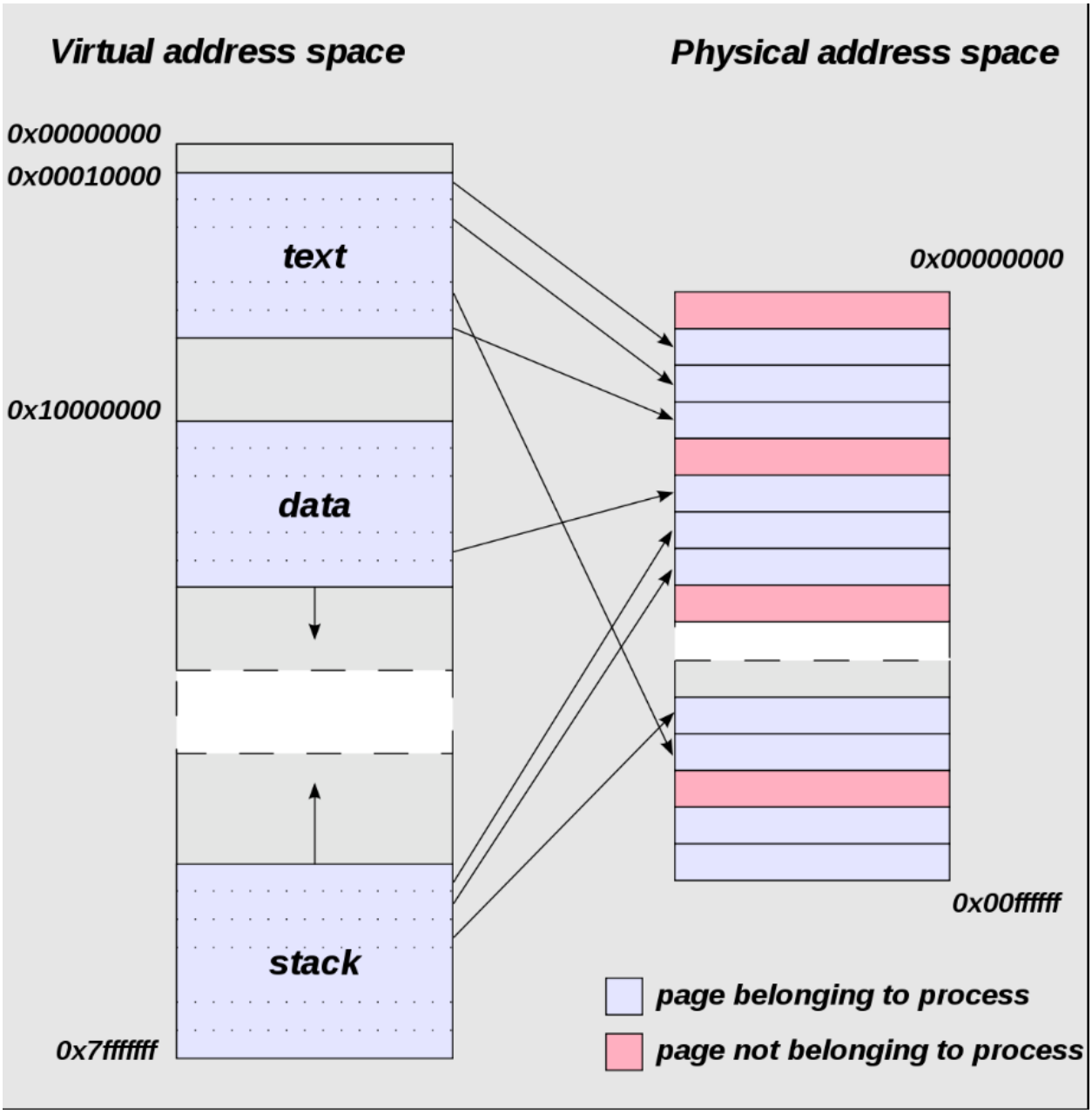


Q: How much memory can you access with a 32 bit address?

- 2^{30} bytes = 1GB
- 2^{32} bytes = 4GB
- 2^{32} words = 16GB

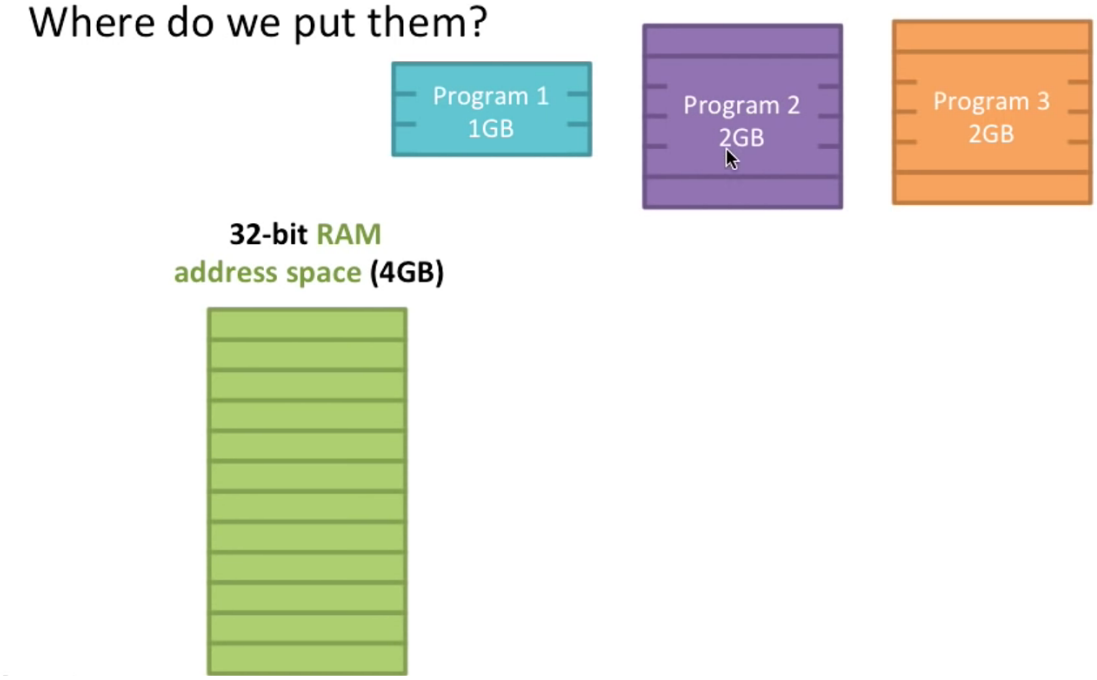
A: 2^{32} bytes = 4GB

A 32-bit address space gives you (theoretically) 4GB of memory you can address. In practice the OS reserves some if it so it is closer to 2GB of usable space.


Virtual address space
Physical address space
0x00000000
0x00010000
text
0x10000000
data
stack
0x7fffffff
0x00000000
0x00ffffff
page belonging to process
page not belonging to process

#2: Holes in our address space

- How do programs share the memory?
- Where do we put them?



#2: Holes in our address space

- How do programs share the memory?
- Where do we put them?

32-bit RAM address space (4GB)

1. Programs 1 and 2 fit
(They use 3GB of memory, leaving **1GB free**.)

#2: Holes in our address space

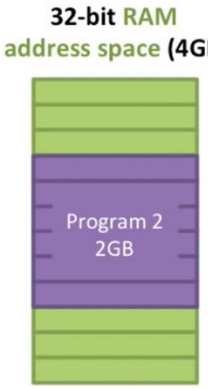
- How do programs share the memory?
- Where do we put them?

32-bit RAM address space (4GB)

1. Programs 1 and 2 fit
(They use 3GB of memory, leaving **1GB free**.)
2. Quit Program 1
(Program 2 uses 2GB of memory, leaving **2GB free**.)

#2: Holes in our address space

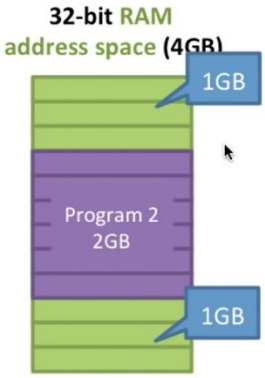
- How do programs share the memory?
- Where do we put them?



1. Programs 1 and 2 fit
(They use 3GB of memory, leaving **1GB free**.)
2. Quit Program 1
(Program 2 uses 2GB of memory, leaving **2GB free**.)
3. Can't run Program 3
(Even though we have enough free space!)

#2: Holes in our address space

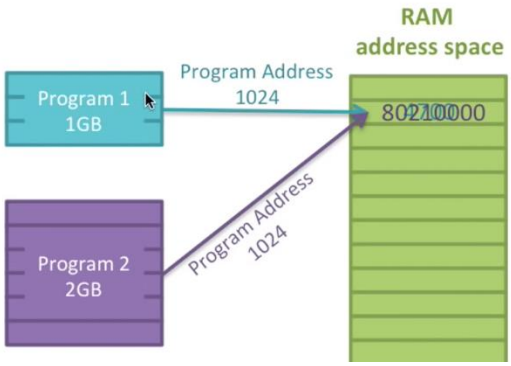
- How do programs share the memory?
- Where do we put them?



1. Programs 1 and 2 fit
(They use 3GB of memory, leaving **1GB free**.)
2. Quit Program 1
(Program 2 uses 2GB of memory, leaving **2GB free**.)
3. Can't run Program 3
(Even though we have enough free space!)
Memory fragmentation

#3: How do we keep programs secure?

- Each program can access any 32-bit memory address
- What if multiple programs access the same address?
  - `sw R2, 1024(R0)` will write to address 1024 regardless of the program that's running



1. **Program 1** stores your bank balance at address 1024
2. **Program 2** stores your video game score at address 1204
3. **Profit? Loss? Not good!**

Problems with memory

- If all programs have access to the same 32-bit memory space:
  - Can **crash** if less than 4GB of RAM memory in the system
  - Can **run out of space** if we run multiple programs
  - Can **corrupt** other programs' data

- How do we solve this?
  - Key to the problem: "**same memory space**"
  - Can we give each program it's **own *virtual* memory space**?
  - If so, we can:
    - Separately **map** each **program's memory space** to the **RAM memory** space

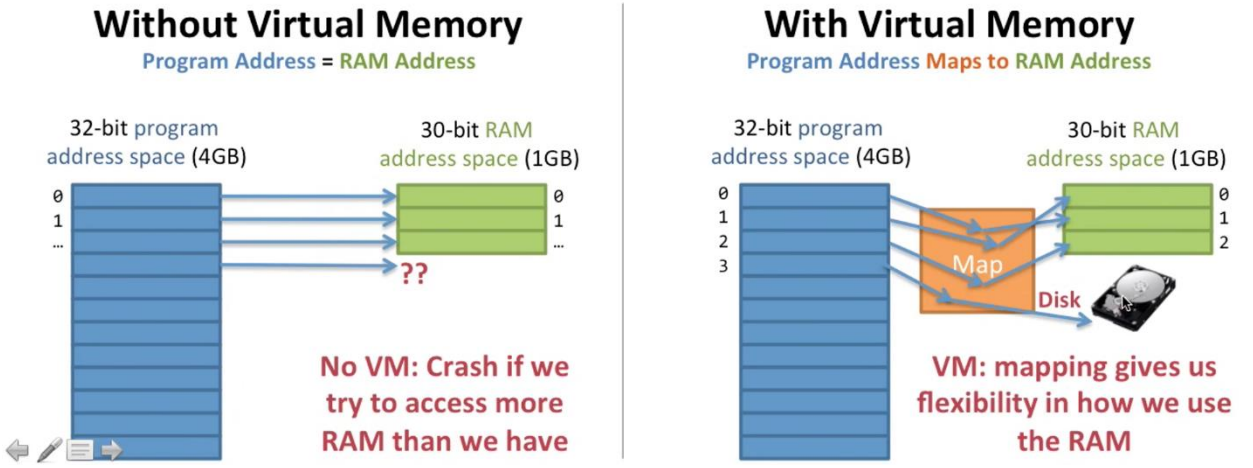
Virtual Memory: 3 What is Virtual Memory?

https://www.youtube.com/watch?v=qlH4-oHnBb8&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=3

Virtual memory is a layer of indirection

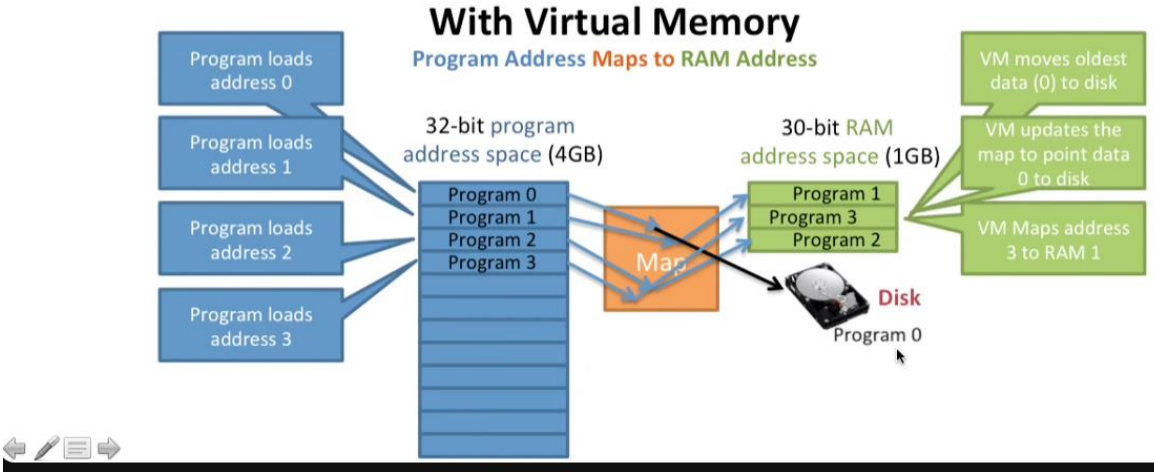
"Any problem in computer science can be solved by adding indirection."

Virtual memory takes **program addresses** and **maps** them to **RAM addresses**



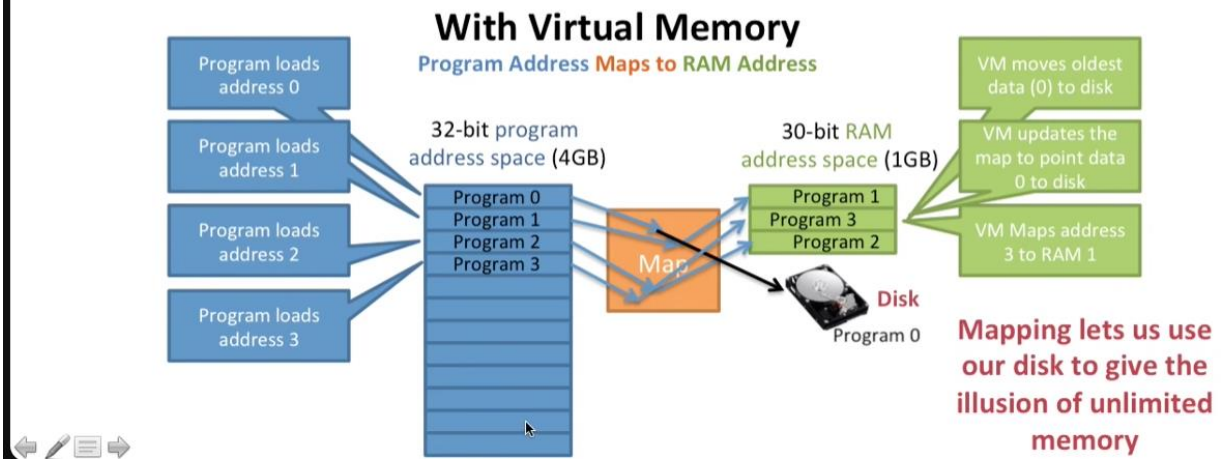
Solving the problems: #1 not enough memory

- **Map** some of the **program's address space** to the **disk**
- When we need it, we bring it into memory



Solving the problems: #1 not enough memory

- **Map** some of the **program's address space** to the **disk**
- When we need it, we bring it into memory



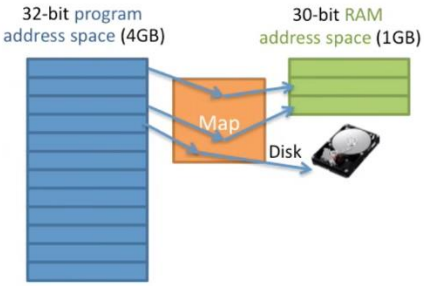
Question: VM and performance

Q: What is going to happen to the program performance when the data it needs is on the disk and not in memory?

- Better performance: we can use more memory than we have
- Nothing: mapping to memory or disk is just as easy
- Worse performance: reading from disk is slower than RAM

A: Worse performance: reading from disk is slower than RAM

Remember that disks are 1000x slower than RAM. Any time you can't fit your data in memory and have to go to disk you pay a HUGE performance penalty! (This is why buying more RAM makes your computer faster.)

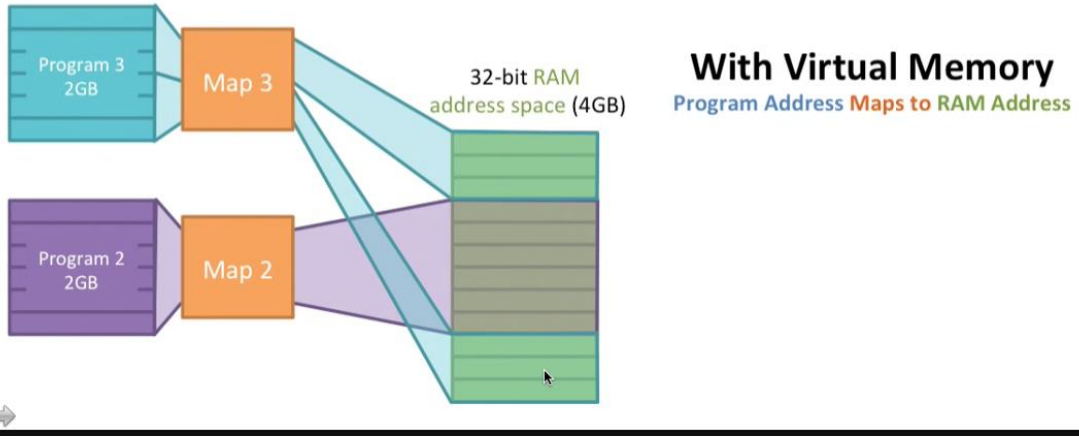


Solving the problems: #2 holes in the address space

- How do we use the holes left when programs quit?
- We can **map** a program's addresses to RAM addresses however we like

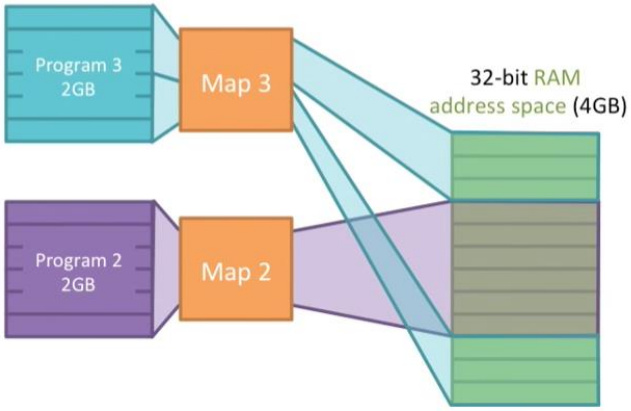
Solving the problems: #2 holes in the address space

- How do we use the holes left when programs quit?
- We can **map** a program's addresses to RAM addresses however we like



Solving the problems: #2 holes in the address space

- How do we use the holes left when programs quit?
- We can **map** a program's addresses to RAM addresses however we like



With Virtual Memory

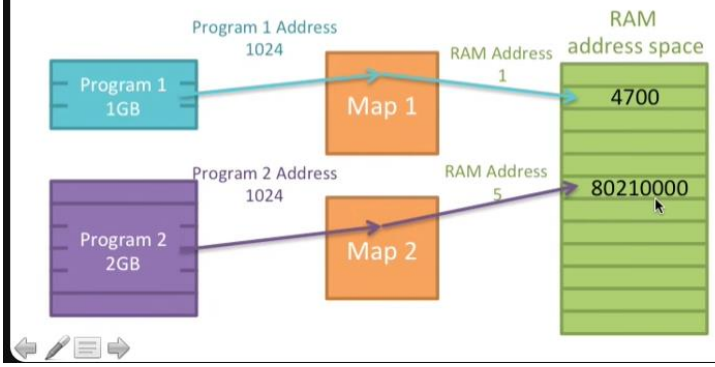
Program Address Maps to RAM Address

Each program has its own mapping.

Mappings lets us put our program data wherever we want in the RAM.

Solving the problems: #3 keeping programs secure

- **Program 1's** and **Program 2's** addresses **map to** different **RAM addresses**



With Virtual Memory

Program Address Maps to RAM Address

1. **Program 1** stores your bank balance at address 1024
1B. **VM maps** it to **RAM address 1**

2. **Program 2** stores your video game score at address 1204
2B. **VM maps** it to **RAM address 5**

Question: Is program isolation always good?

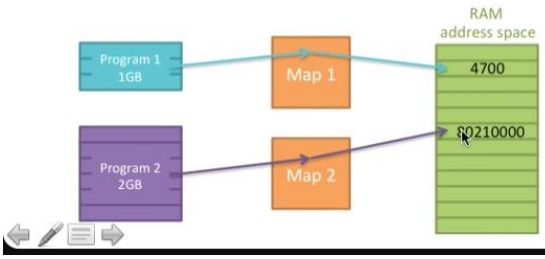
Q: Virtual Memory lets us isolate programs so they can't share/corrupt data. What is a downside of complete isolation?

- Programs can't corrupt each other
- Programs can't share data with each other
- Programs use more space because they have their own address space
- Programs are slower because they always have to check the disk for data

A: Programs can't share data with each other

A lot of data is shared between programs. (Think about shared resources: fonts, graphics, scrollbars, icons and shared functionality: libraries, open/save dialog boxes, etc.)

However, we can use the same mapping to allow programs to share data by simply having their maps point to the same data! (Isn't indirection great?)



Virtual Memory: 4 How Does Virtual Memory Work?

https://www.youtube.com/watch?v=59rEMnKWoS4&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=4

How does VM work?

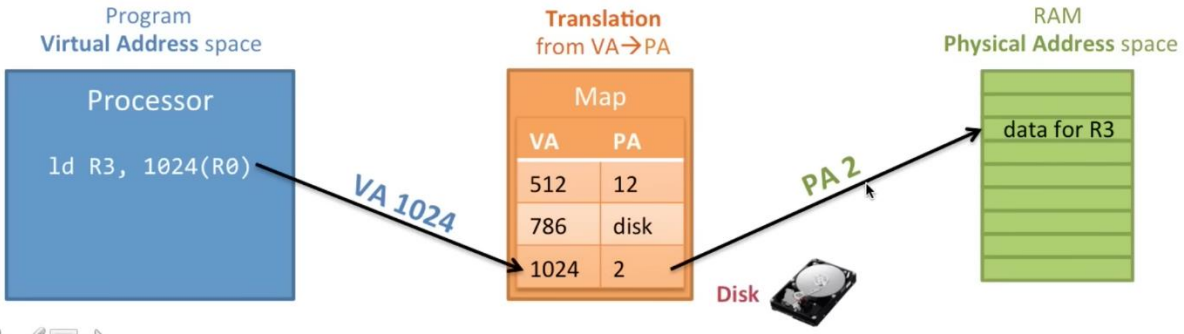
- **Basic idea: separate memory spaces**
  - **Virtual memory**: what the **program** sees
    - e.g., ld R4, 1024(R0) accesses virtual address R0+1024=1024
  - **Physical memory**: the **physical RAM** in the computer
    - e.g., if you have 2GB of RAM installed, you have physical addresses 0 to 2^{31}-1

- **Virtual Addresses (VA)**
  - What the program uses
  - In MIPS, this is the full 32-bit address space: 0 to 2^{32}-1
- **Physical Addresses (PA)**
  - What the hardware uses to talk to the RAM
  - Address space determined by how much RAM is installed

Making VM work: translation

How does a program access memory?

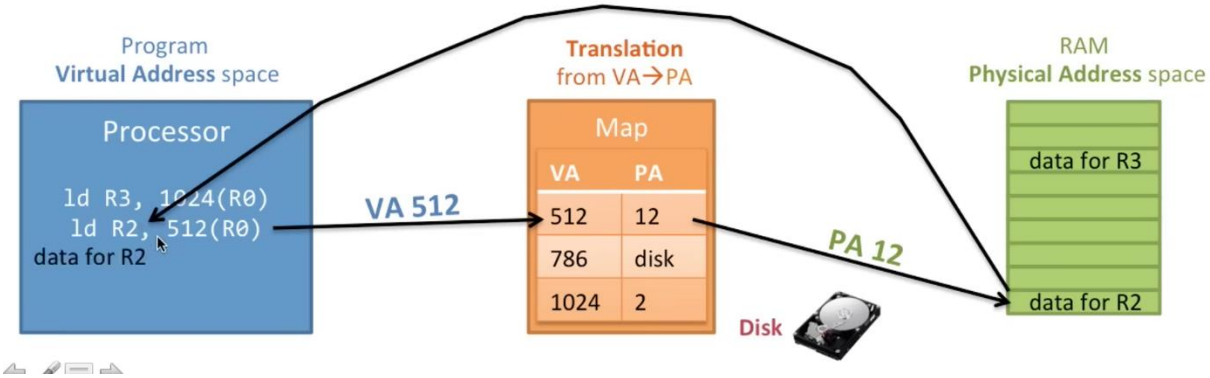
1. Program executes a load specifying a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system **loads it in from disk**)
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to the program



Making VM work: translation

How does a program access memory?

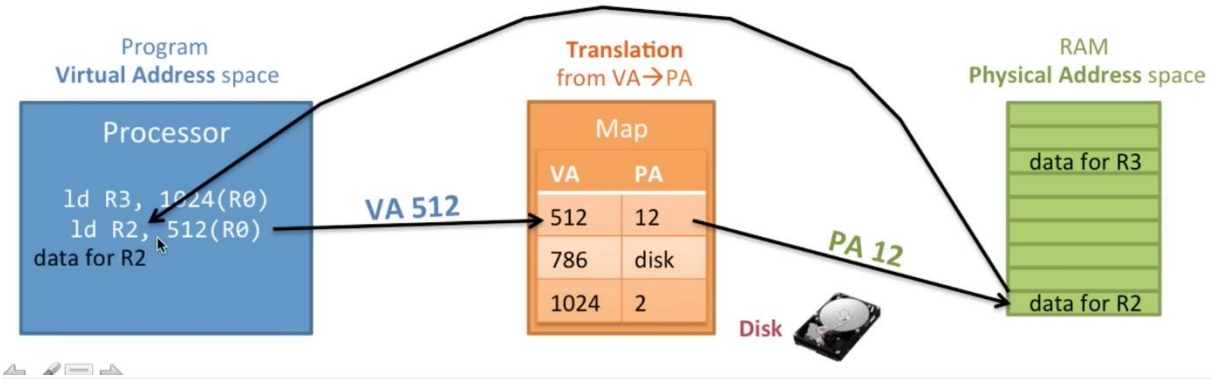
1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system **loads it in from disk**)
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to the program



Making VM work: translation

How does a program access memory?

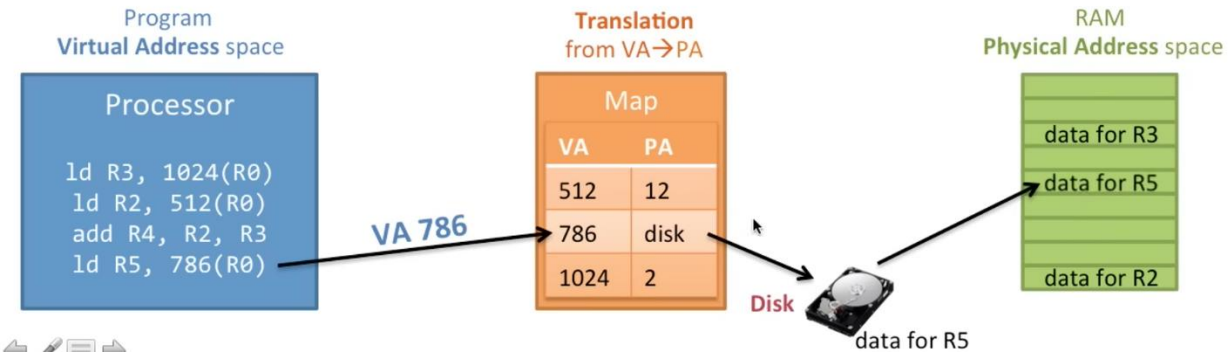
1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system **loads it in from disk**)
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to the program



Making VM work: translation

How does a program access memory?

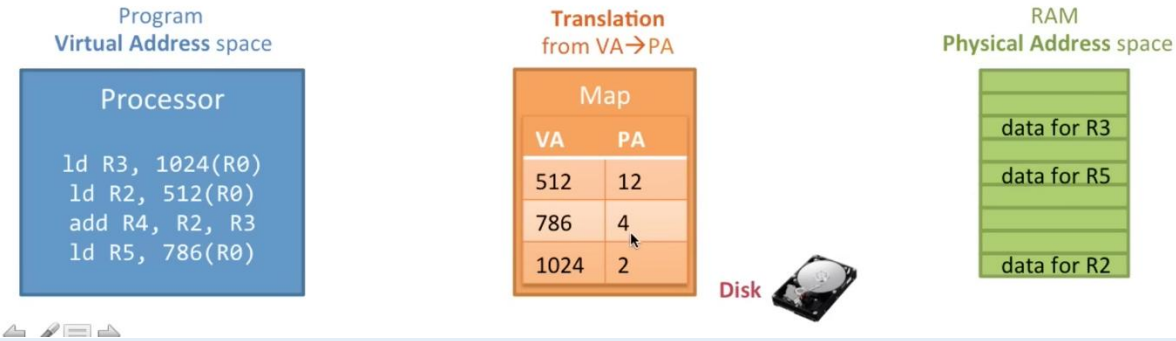
1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system **loads it in from disk**)
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to th program



Making VM work: translation

How does a program access memory?

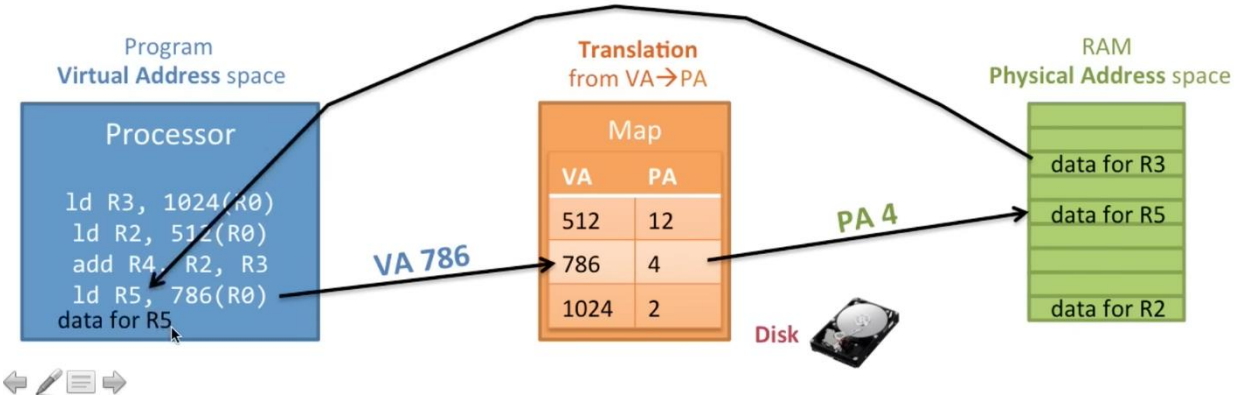
1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system loads it in from **disk)**
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to the program



Making VM work: translation

How does a program access memory?

1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system loads it in from **disk)**
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to th program



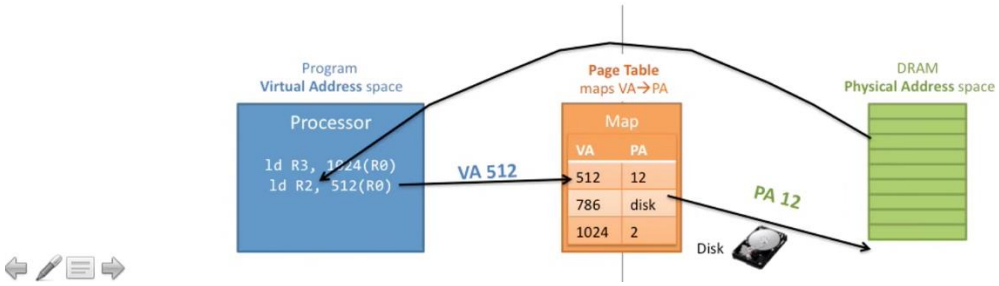
Virtual Memory: 5 Page Tables

https://www.youtube.com/watch?v=KNUJhZCQZ9c&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=5

Page tables

The map from Virtual Addresses (VA) to Physical Addresses (PA) is the Page Table

So far we have had one Page Table Entry (PTE) for every Virtual Address



Page tables

The map from Virtual Addresses (VA) to Physical Addresses (PA) is the Page Table

So far we have had one Page Table Entry (PTE) for every Virtual Address

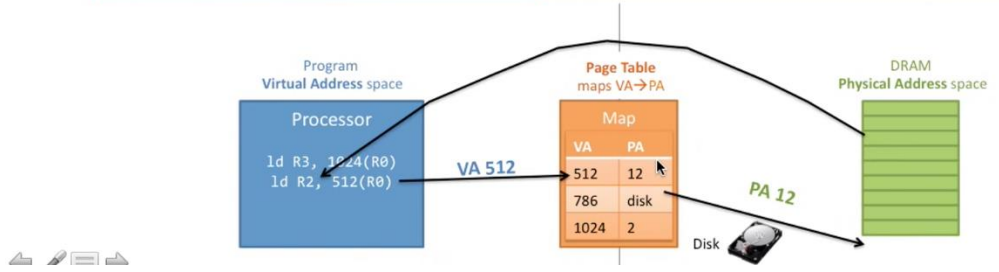
Q: How many entries do we need in our Page Table?

- 1 for every byte = 2^{32} = 4 billion
- 1 for every word = 2^{30} = 1 billion
- 1 for every register = 32

A: 1 for every word = 2^{30} = 1 billion

We have a word-aligned memory so we need to be able to address every word.

Note: each entry needs the PA which is 32 bits, so that's 1GB of memory for the Page Table alone!



Page table size

- We need to translate every possible address:
  - Our programs have 32-bit Virtual Address spaces
  - That's 2^{30} words that need Page Table Entries (1 billion entries!)

  (If they don't have a Page Table Entry then we can't access them because we can't find the physical address.)
- How can we make this more manageable?
  - What if we divided memory up into chunks (**pages**) instead of words?

Fine-grain:
Maps each word address
2^{30} words to map

Page Table
translates VA→PA

Map

VA	PA
512	12
786	3
1024	2

4kB Pages:
Each entry now covers 4kB of data.

Coarse-grain:
Maps chunks of address
Fewer mappings

Page Table
translates VA→PA

Map

VA	PA
0-4095	4096-8191

Coarse-grained: pages instead of words

- The **Page Table** manages larger chunks (**pages**) of data:
  - Fewer Page Table Entries needed to cover the whole address space
  - But, less flexibility in how to use the RAM (have to move a page at a time)
- Today:
  - Typically 4kB pages (1024 words per page)

Q: How many entries do we need in our Page Table with 4kB pages on a 32-bit machine?
- 1 for every word = 2^{30} = 1 billion
- 1 for every 1024 words = 1 million
- 1 for every 4096 words = 262,144

A: 1 for every 1024 words = 1 million
With 4kB pages we have 1024 words per page. That means we need 1 Page Table Entry for every 1024 words. In total we have 1 billion words, so 1 billion ÷ 1024 is 1 million Page Table Entries. This is *much* more manageable.

Coarse-grain:
maps chunks of address

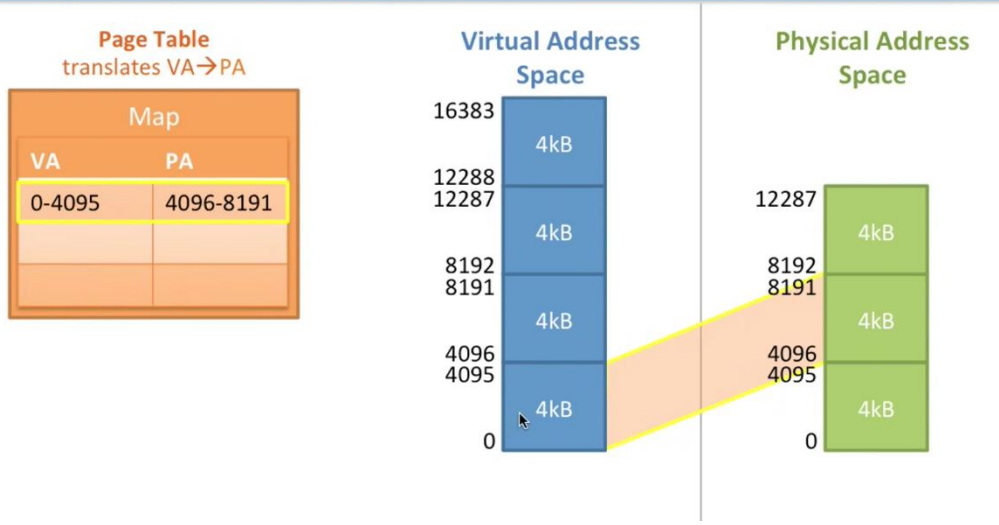
Page Table
translates VA→PA

Map

VA	PA
0-4095	4096-8191

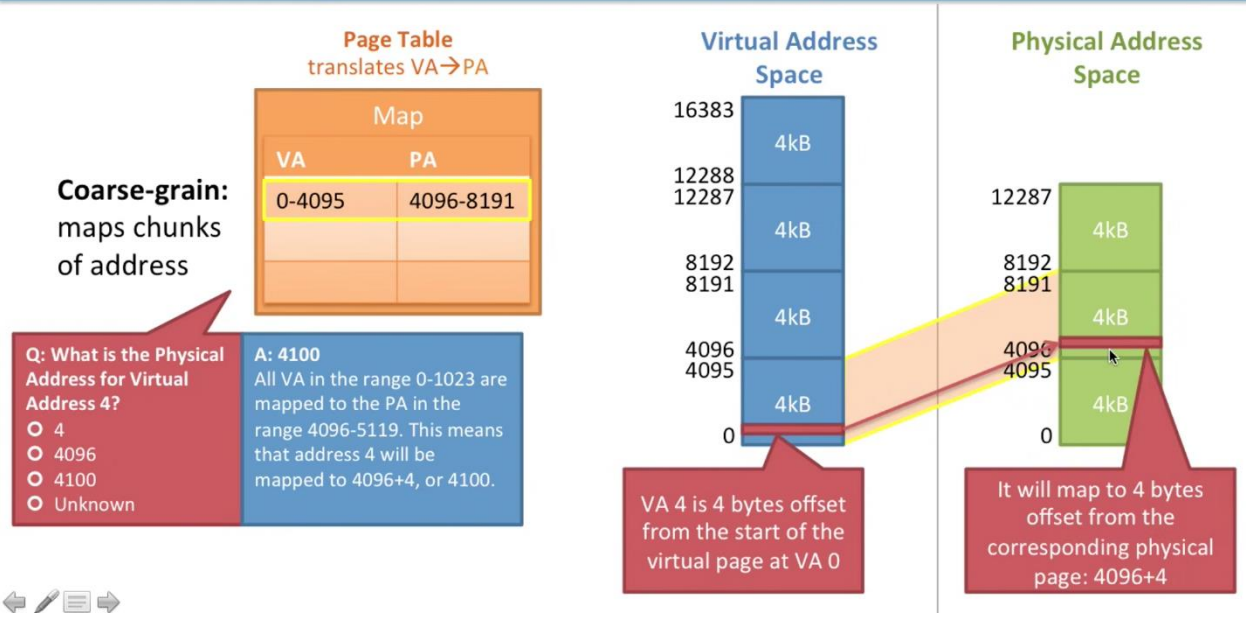
How do we map addresses with pages?

Coarse-grain: maps chunks of address



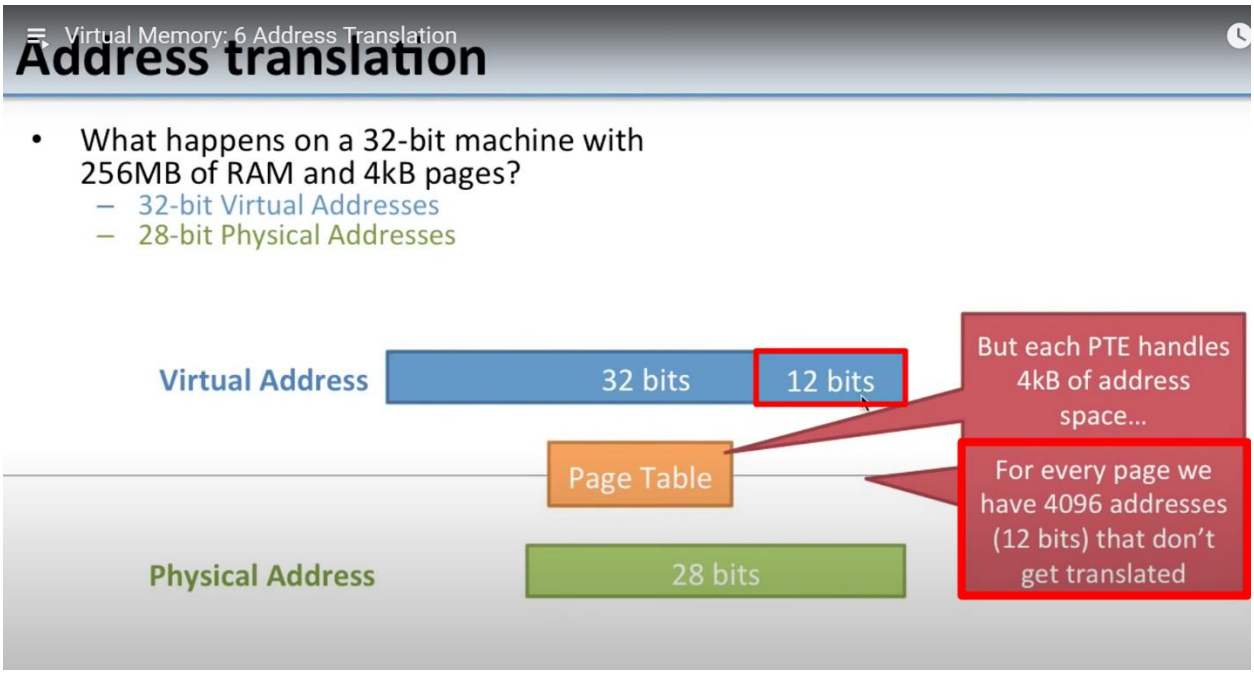
How do we map addresses with pages?

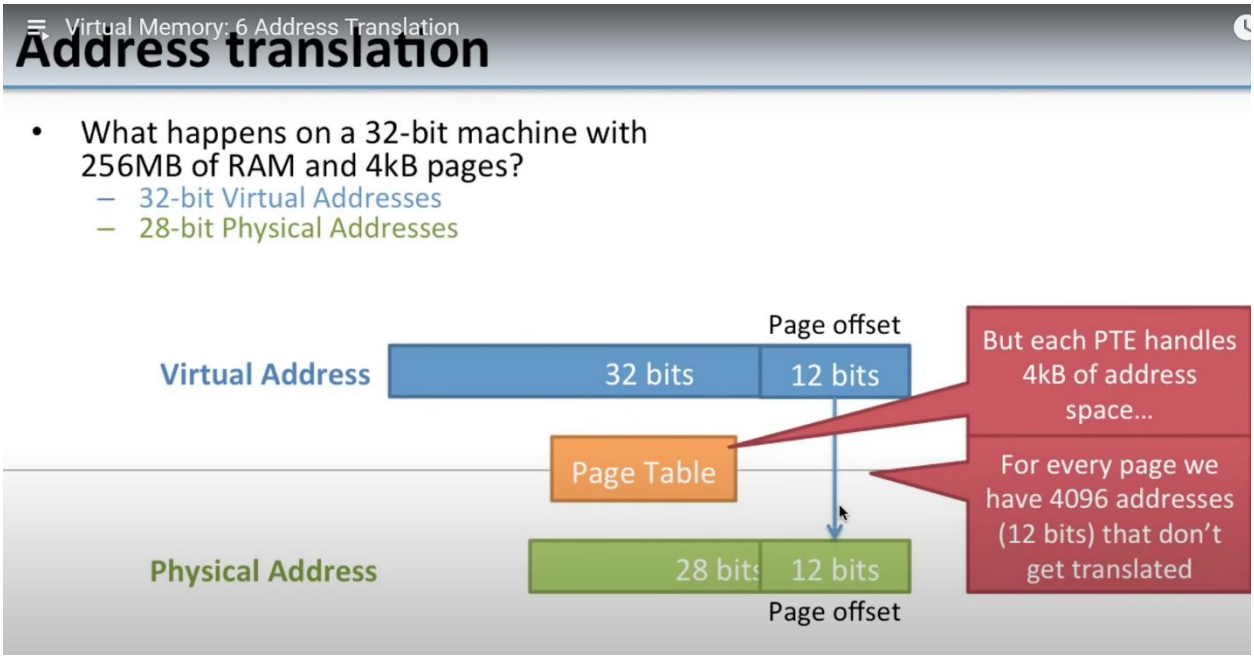
Coarse-grain: maps chunks of address



Virtual Memory: 6 Address Translation

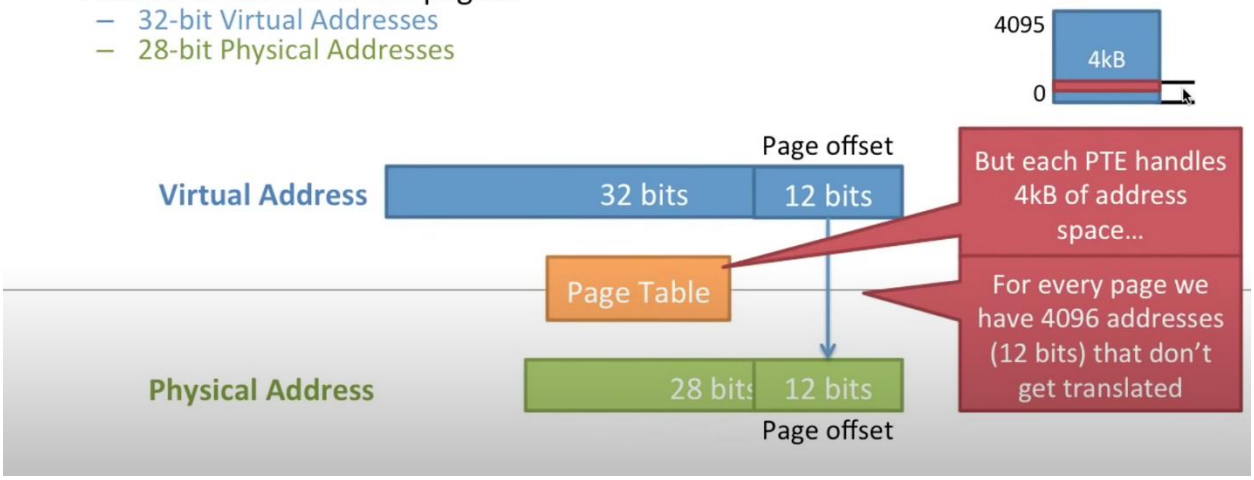
https://www.youtube.com/watch?v=ZjKS1IbiGDA&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=6





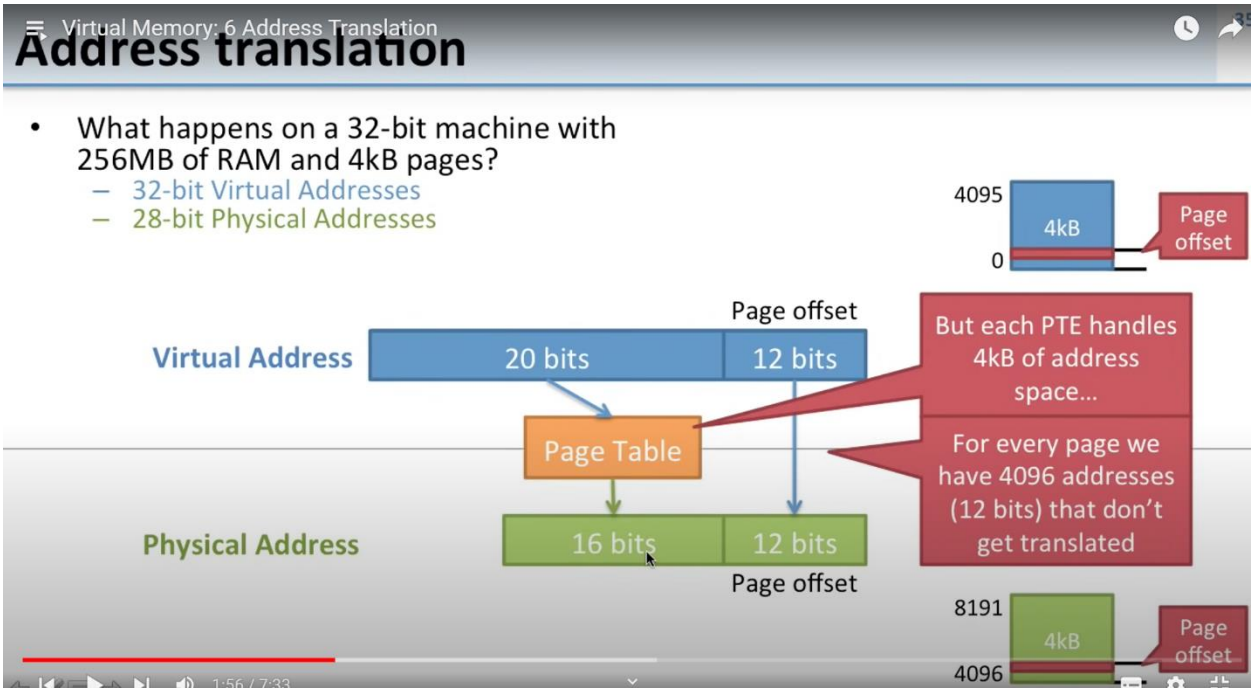
Address translation

- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?
  - 32-bit Virtual Addresses
  - 28-bit Physical Addresses



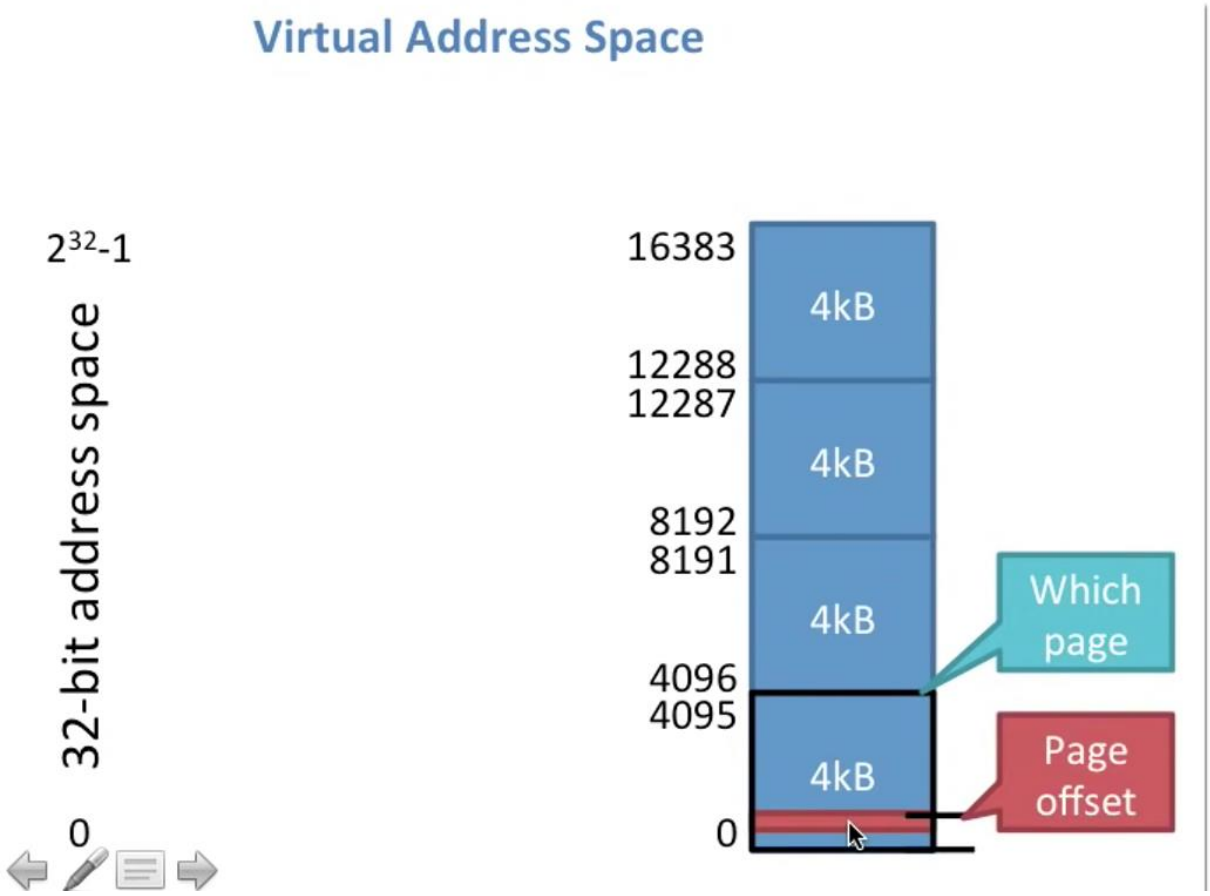
Address translation

- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?
  - 32-bit Virtual Addresses
  - 28-bit Physical Addresses



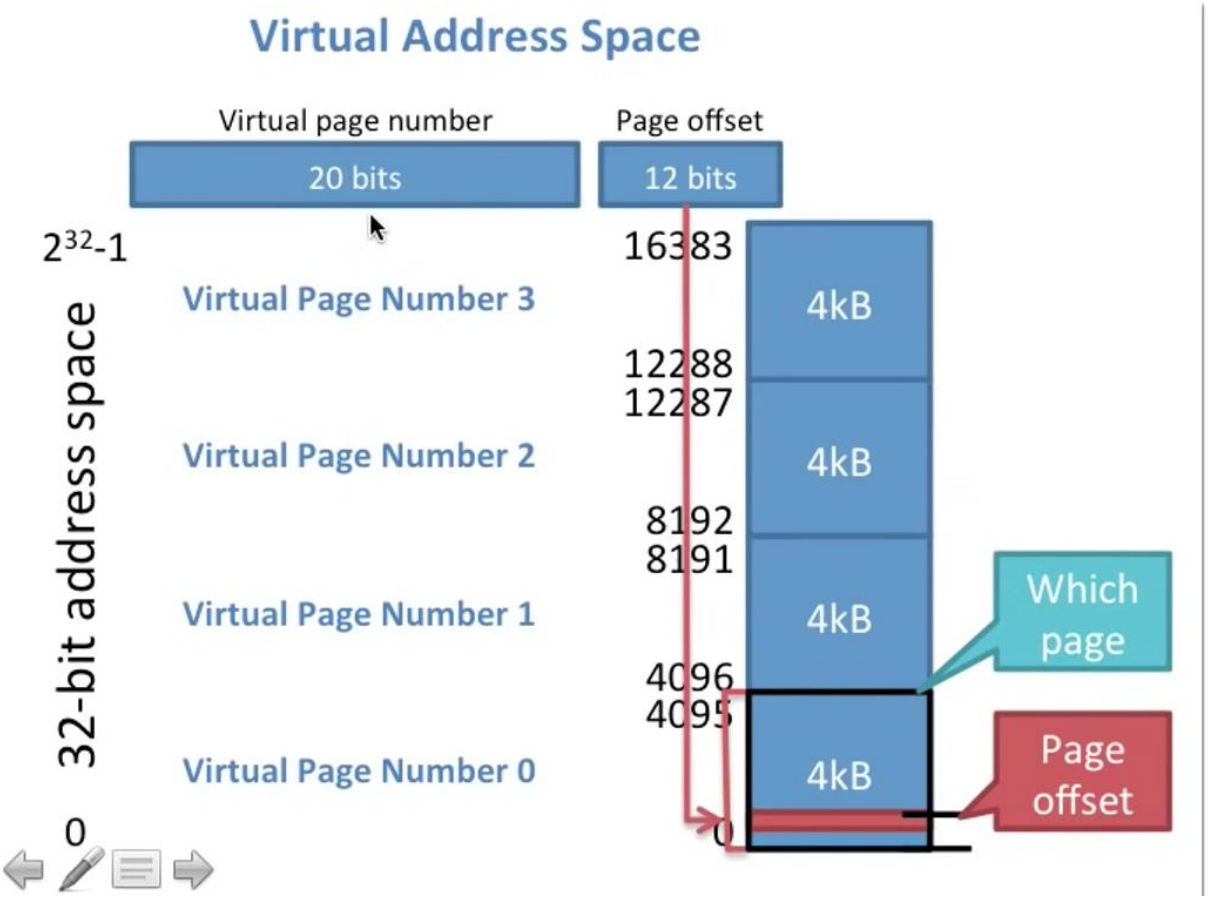
Pages, offsets, and translation

- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?



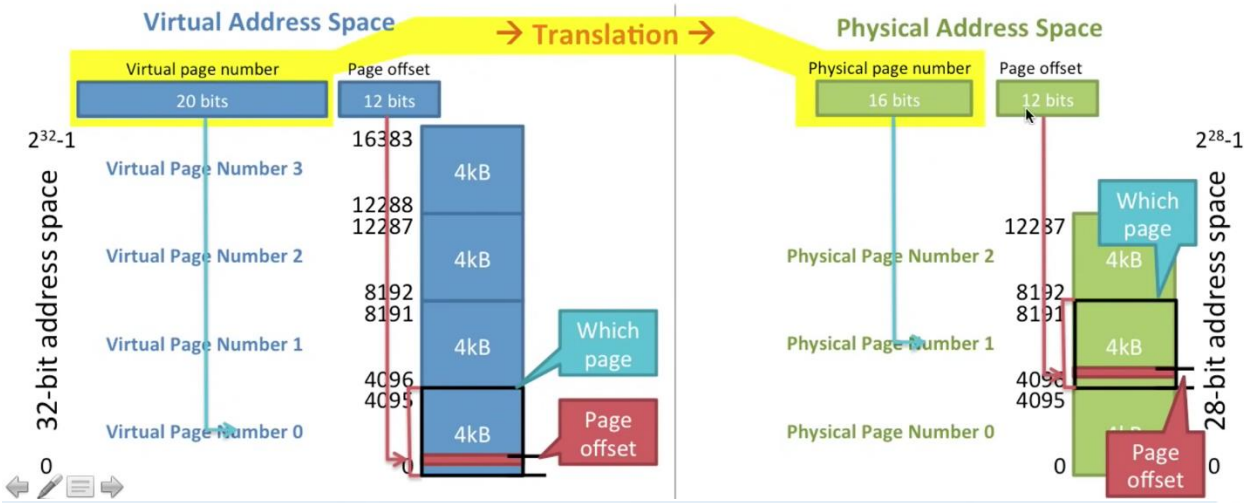
Pages, offsets, and translation

- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?



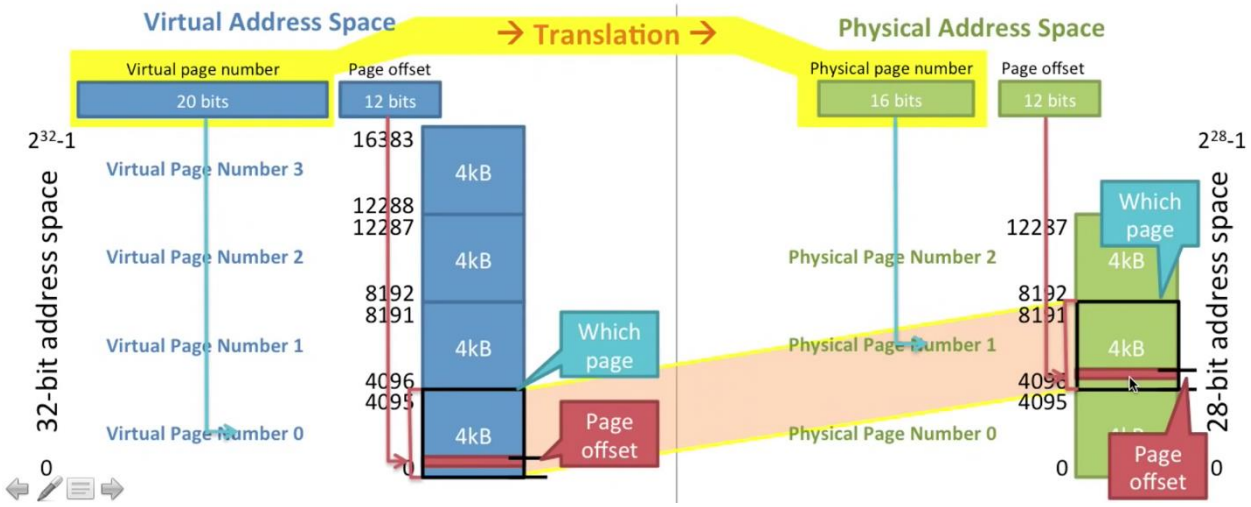
Pages, offsets, and translation

- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?

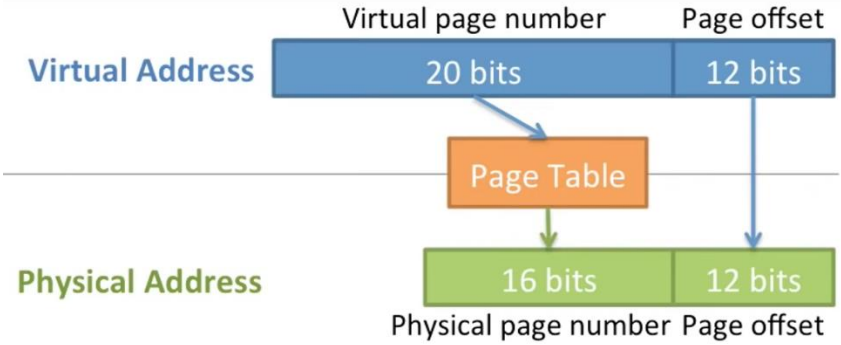


Pages, offsets, and translation

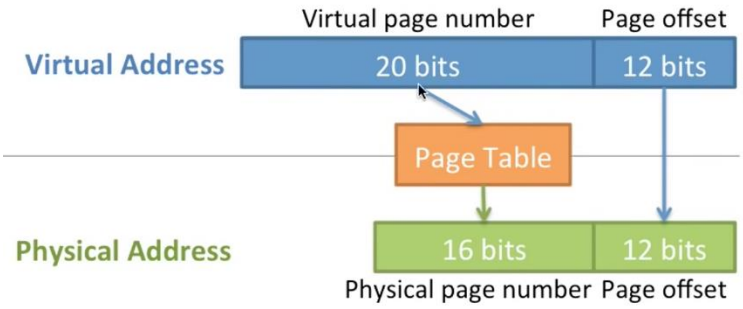
- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?



Question: virtual vs. physical page number bits



Question: virtual vs. physical page number bits



Q: Why do we have more virtual page number bits than physical page number bits in this example?

- The page offset is different for the virtual and physical addresses
- The virtual address space (size) is larger than our physical RAM
- The page table cannot map to a larger number of bits than its input

A: The virtual address space (size) is larger than our physical RAM

We had a 4GB virtual address space (2^{32} bytes) but only a 256MB physical address space (2^{28} bytes). So we needed fewer bits to address our physical address space.

Question: What address is loaded?

Q: What would be true on a 32-bit machine with 8GB of RAM installed?

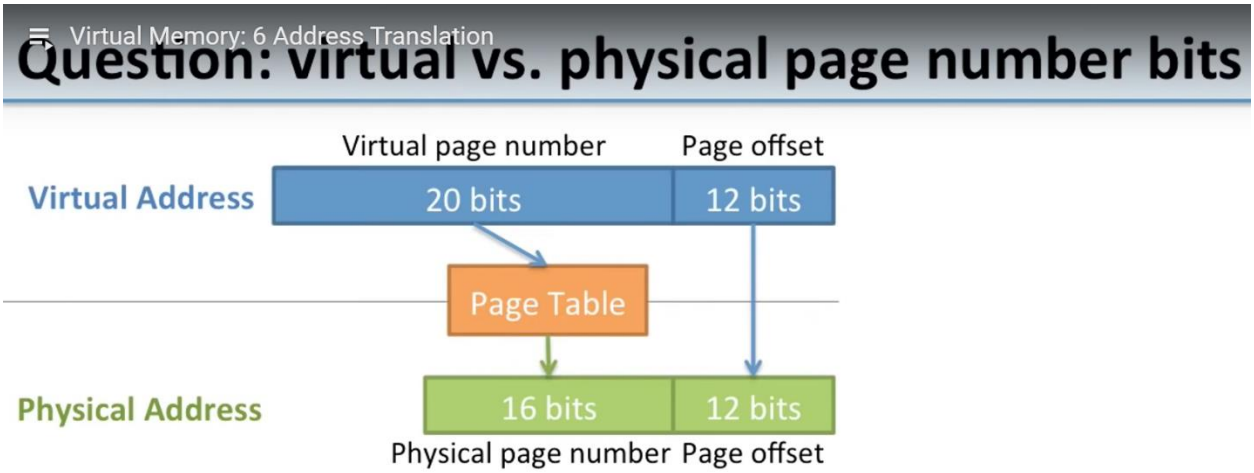
- The page offset would be larger
- The physical address would be the same size as the virtual address
- You would not need virtual memory
- The physical address would be larger than the virtual address

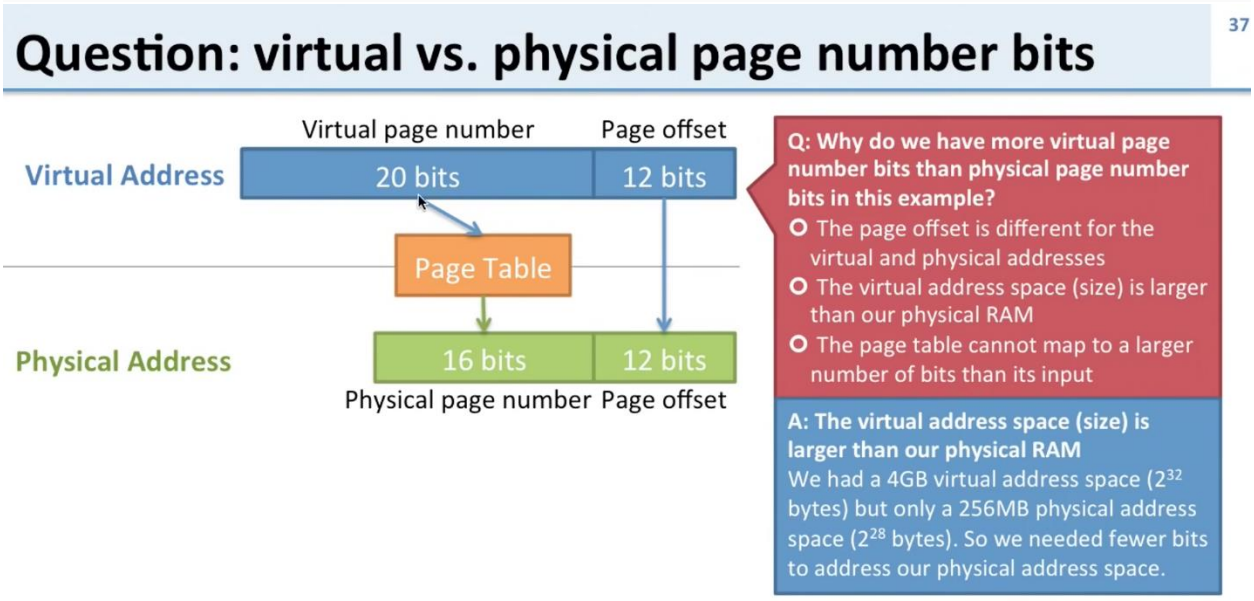
A: The physical address would be larger than the virtual address

To address 8GB of memory you need 33 bits of address. This means the physical address will need to be 33 bits. Of course each program can only address 32 bits because the program address space is only 32 bits. (This is why we have moved to 64 bit processors.)

Virtual Memory: 7 Address Translation Example Walkthrough

https://www.youtube.com/watch?v=6neHHkI0Z0o&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=7





Question: What address is loaded?

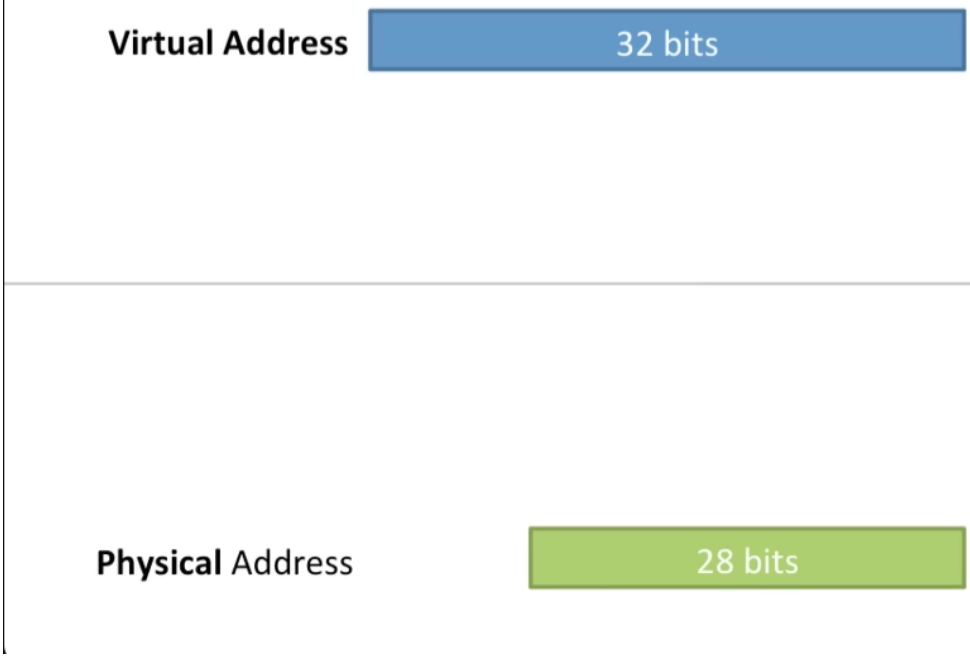
Q: What would be true on a 32-bit machine with 8GB of RAM installed?

- The page offset would be larger
- The physical address would be the same size as the virtual address
- You would not need virtual memory
- The physical address would be larger than the virtual address

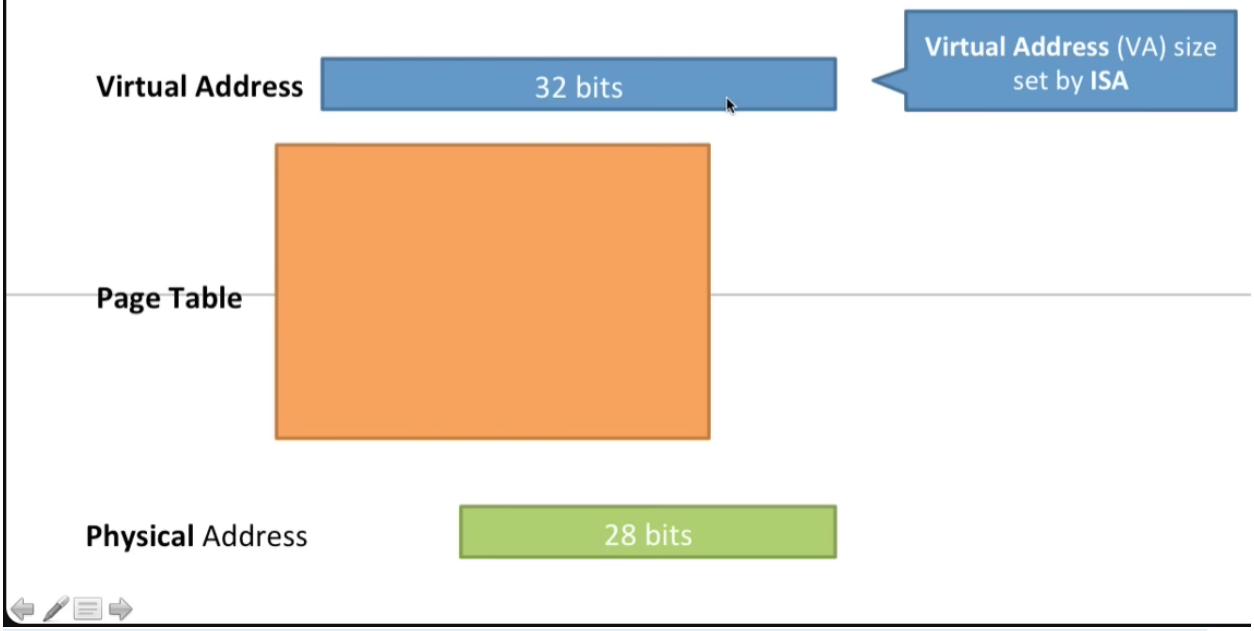
A: The physical address would be larger than the virtual address

To address 8GB of memory you need 33 bits of address. This means the physical address will need to be 33 bits. Of course each program can only address 32 bits because the program address space is only 32 bits. (This is why we have moved to 64 bit processors.)

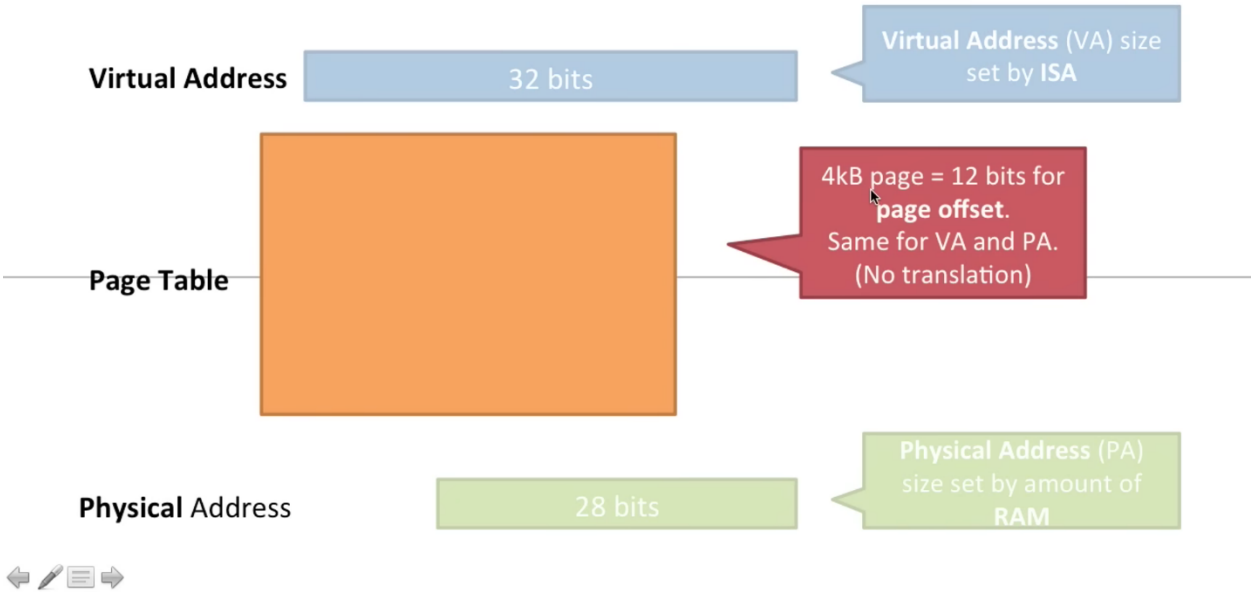
How to do a page table lookup



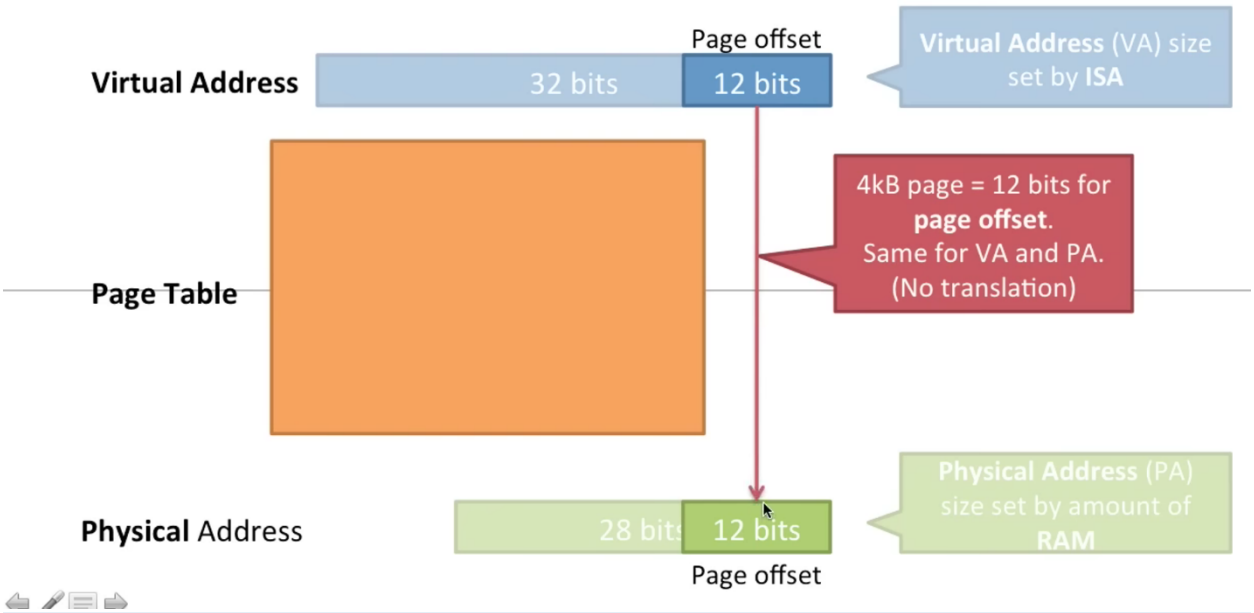
How to do a page table lookup



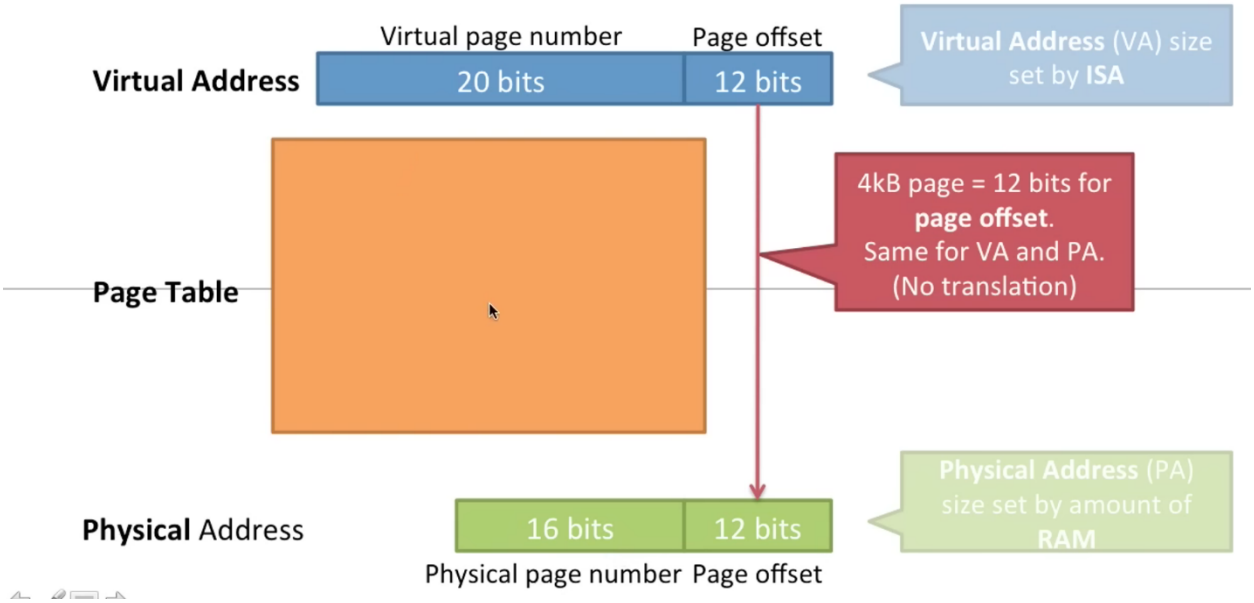
How to do a page table lookup

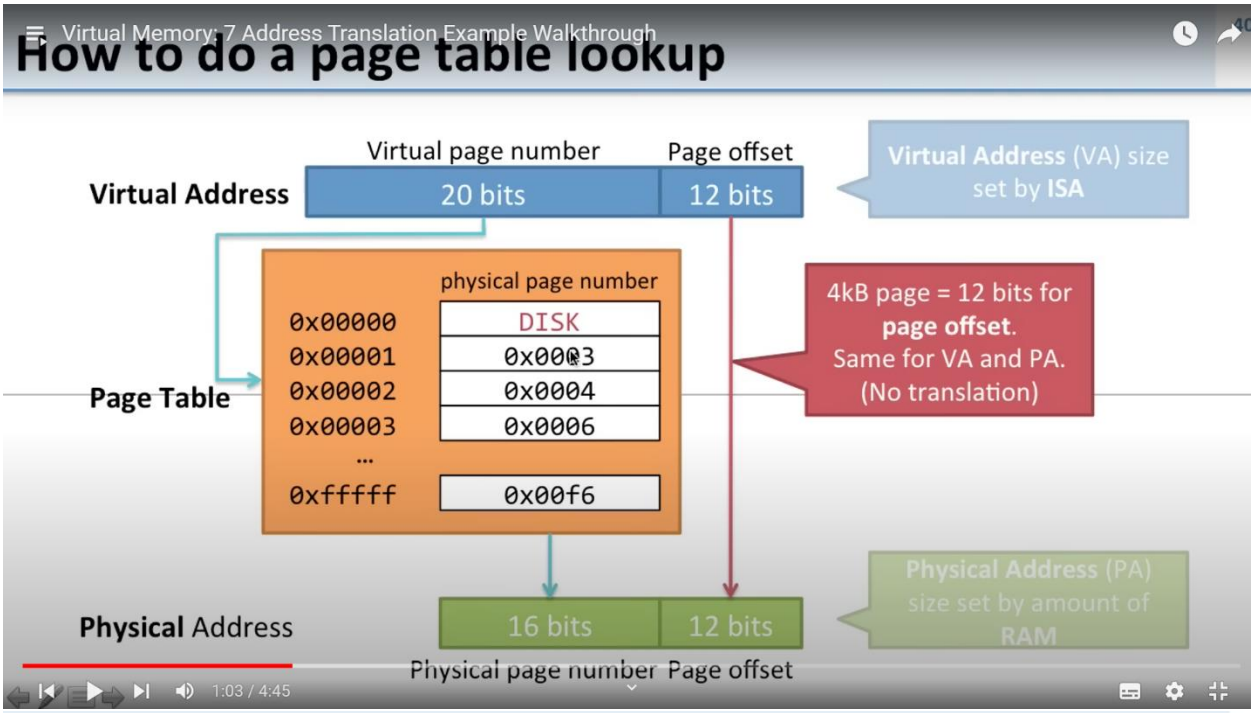


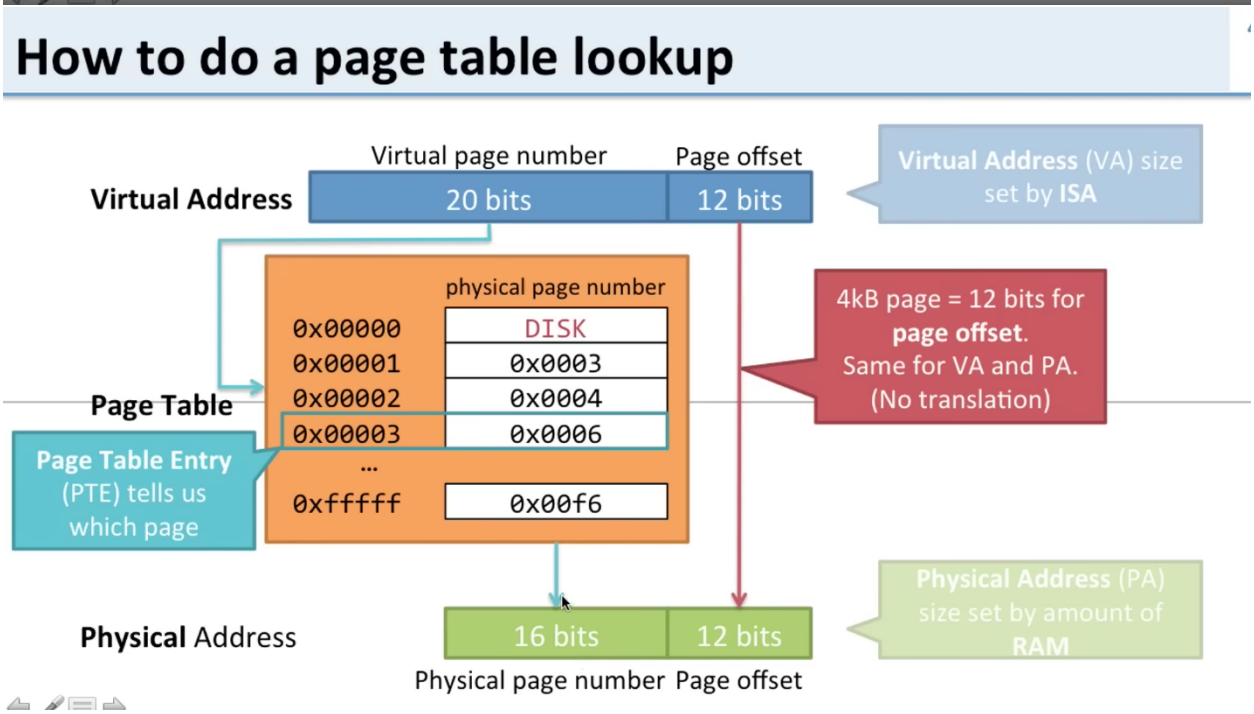
How to do a page table lookup



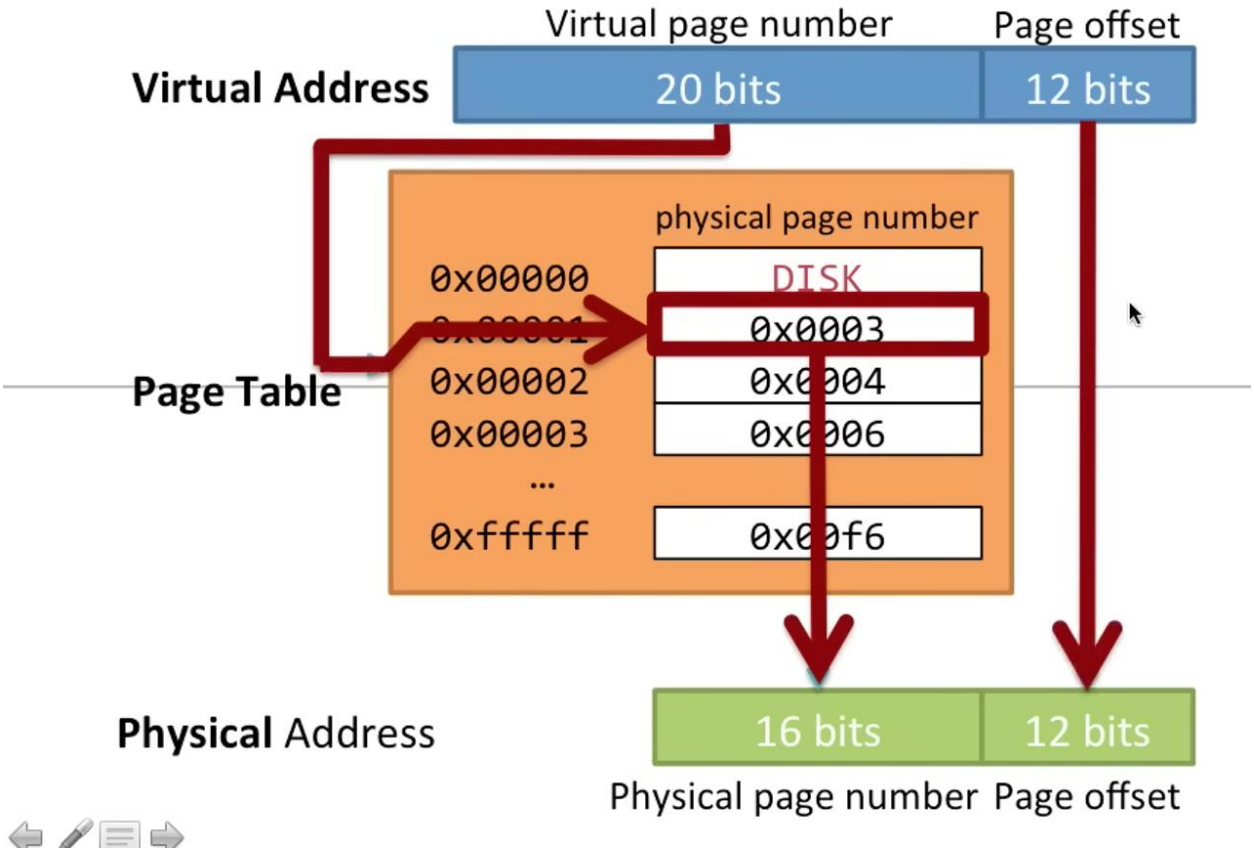
How to do a page table lookup



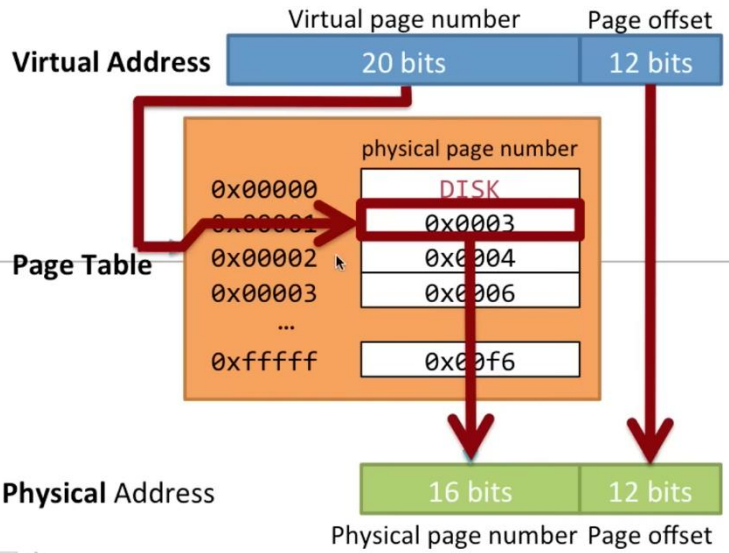

Virtual Memory: 7 Address Translation Example Walkthrough
How to do a page table lookup
Virtual page number
Page offset
Virtual Address
20 bits
12 bits
Virtual Address (VA) size set by ISA
physical page number
0x00000
DISK
0x00001
0x0003
0x00002
0x0004
0x00003
0x0006
...
0xfffff
0x00f6
Page Table
4kB page = 12 bits for page offset. Same for VA and PA. (No translation)
Physical Address
16 bits
12 bits
Physical Address (PA) size set by amount of RAM
Physical page number
Page offset
1:03 / 4:45


How to do a page table lookup
Virtual page number
Page offset
Virtual Address
20 bits
12 bits
Virtual Address (VA) size set by ISA
physical page number
0x00000
DISK
0x00001
0x0003
0x00002
0x0004
0x00003
0x0006
...
0xfffff
0x00f6
Page Table
4kB page = 12 bits for page offset. Same for VA and PA. (No translation)
Page Table Entry (PTE) tells us which page
Physical Address
16 bits
12 bits
Physical Address (PA) size set by amount of RAM
Physical page number
Page offset

How to do a page table lookup

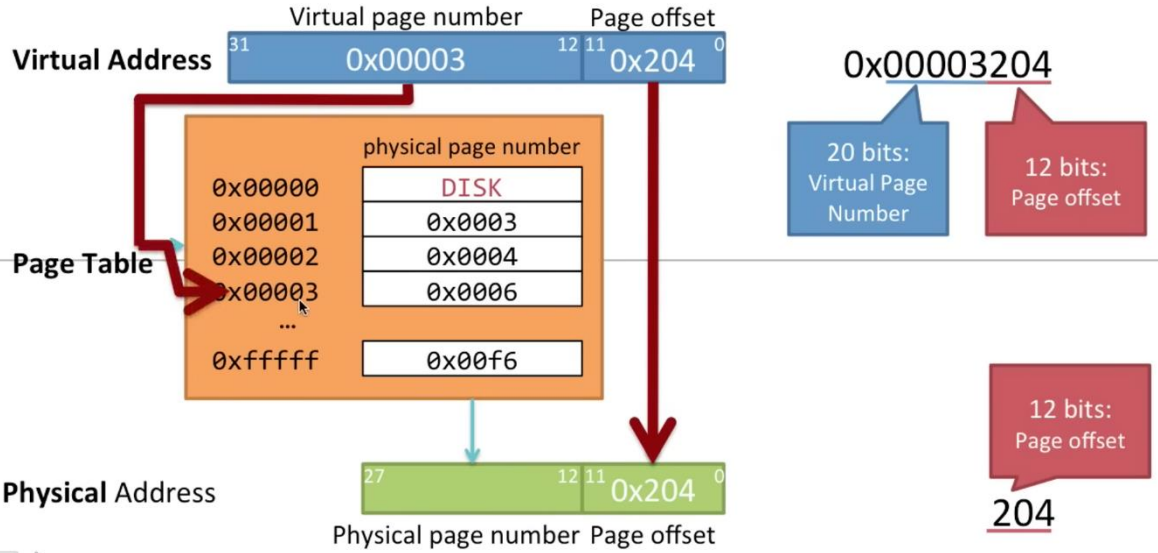


How to do a page table lookup

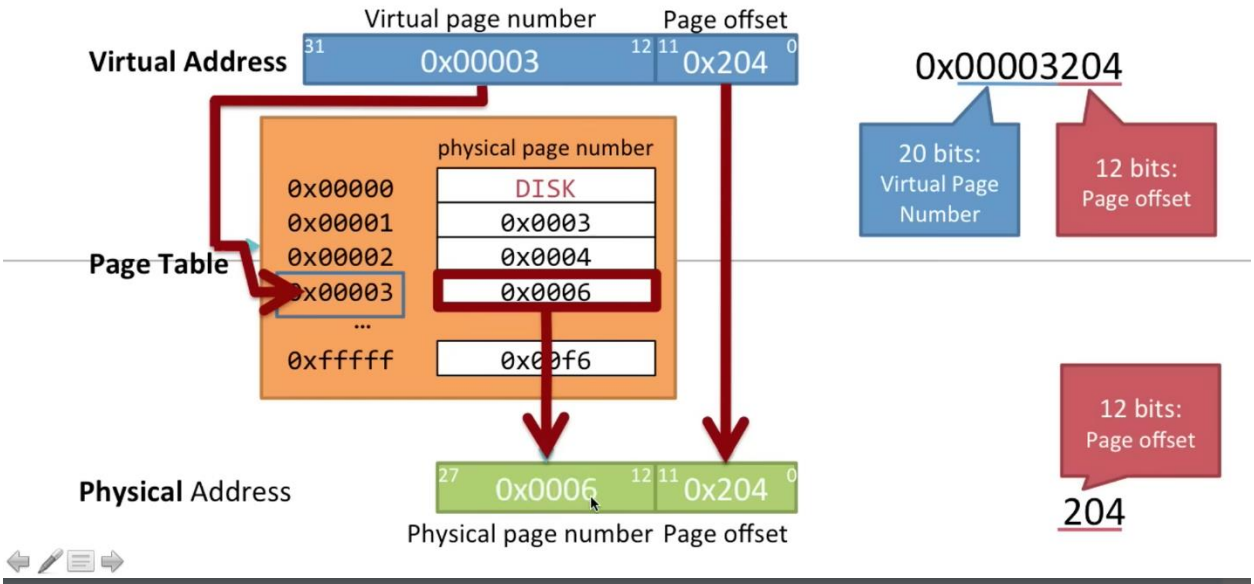


- **Virtual page number translated** to the **physical page number**
- **Page offset** untranslated
- **Page Table** keeps mappings for every virtual page → physical page

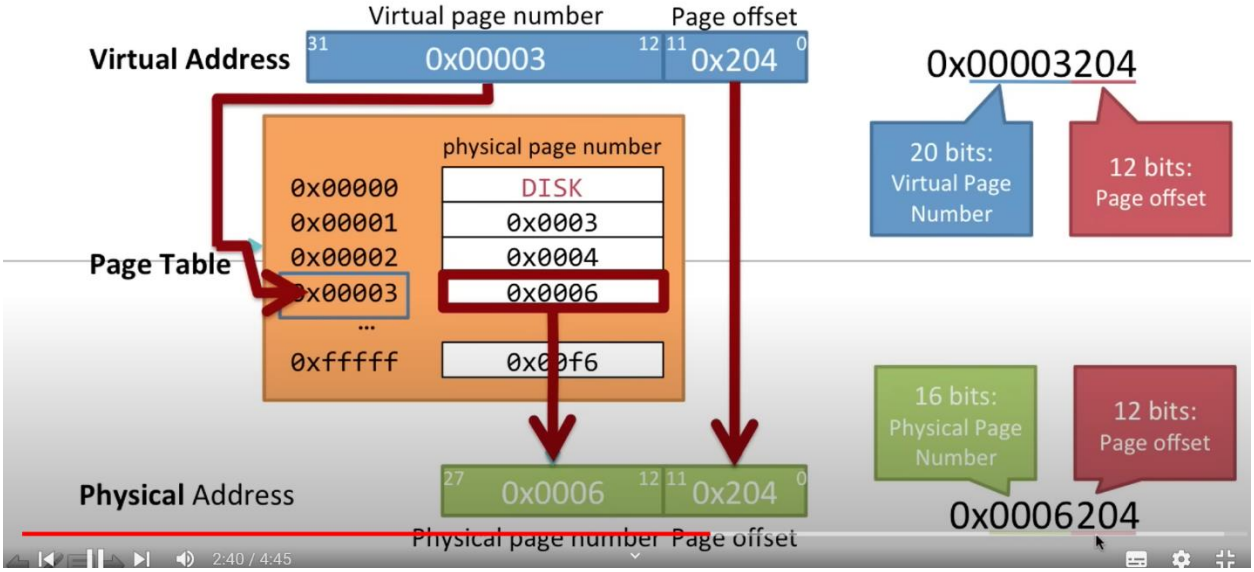
Example translation



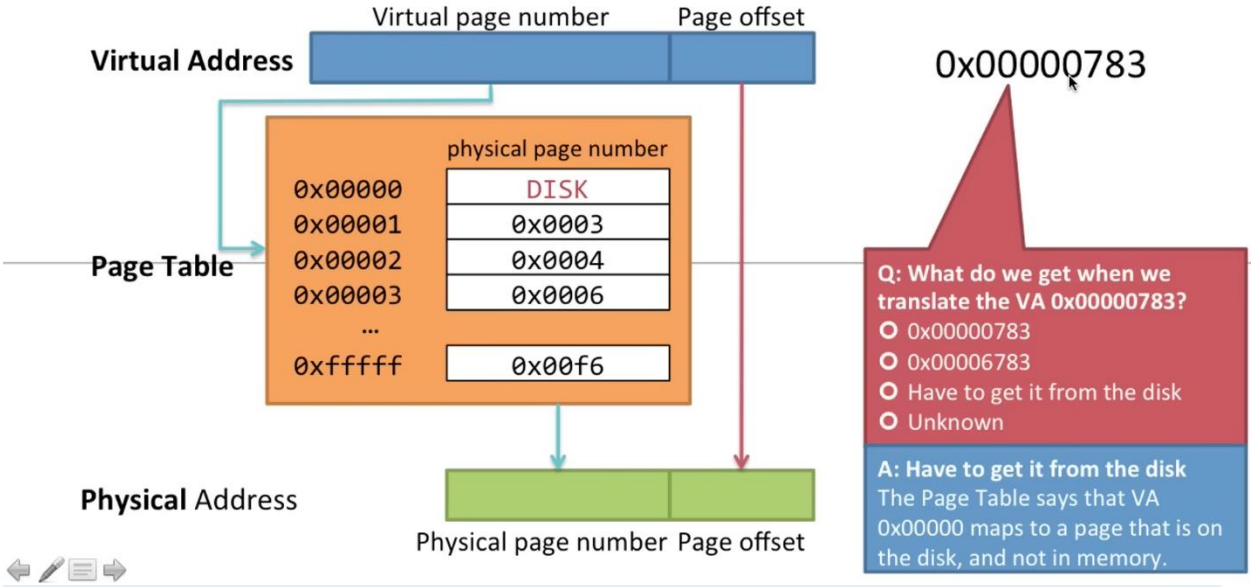
Example translation



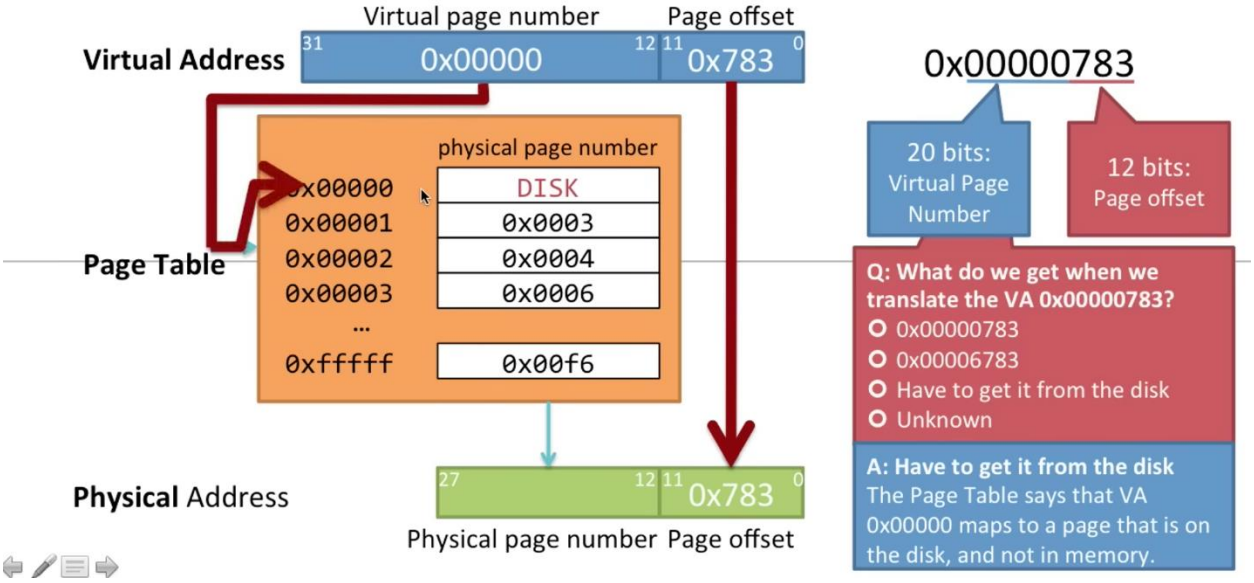
Virtual Memory: 7 Address Translation Example Walkthrough
Example translation



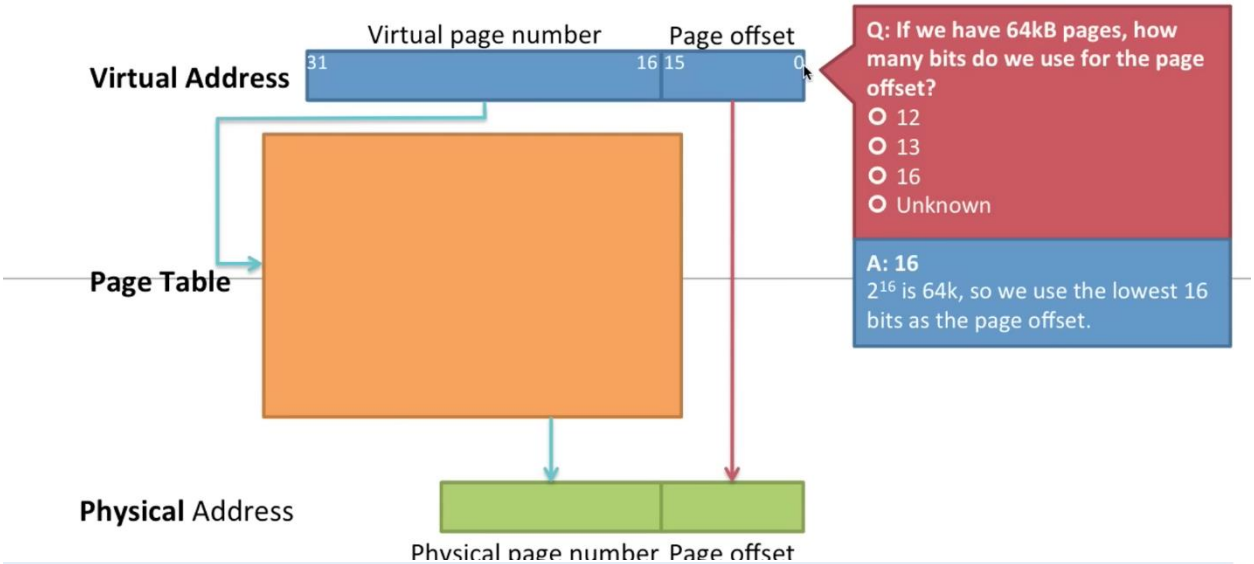
Example translation



Example translation



Example translation: 64kB pages



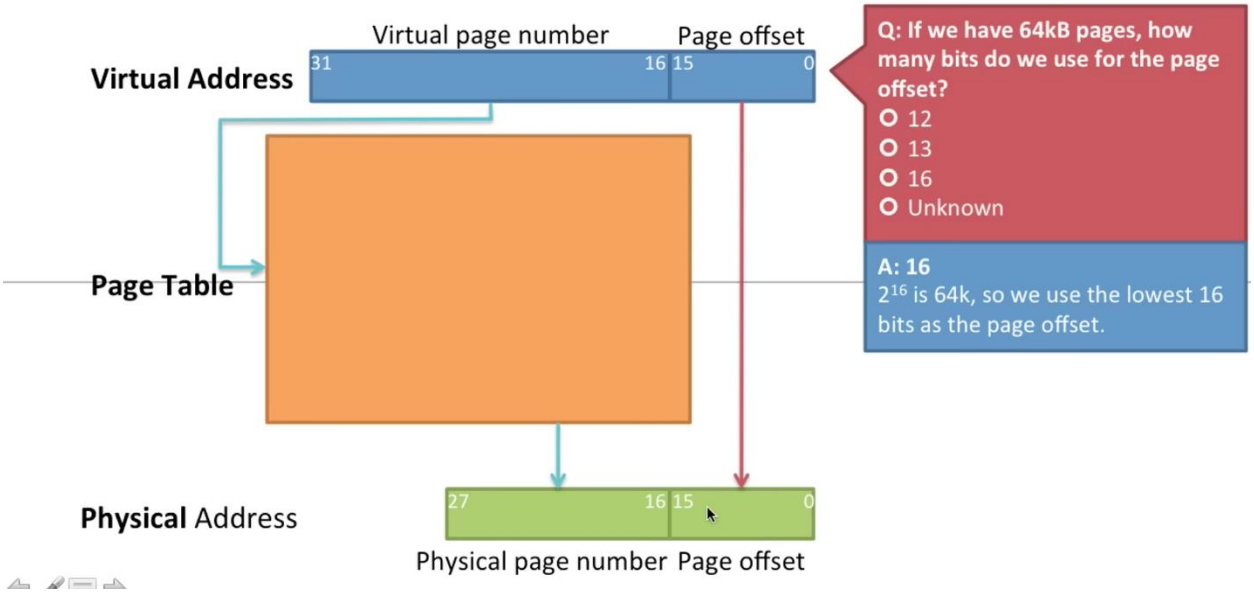
Q: If we have 64kB pages, how many bits do we use for the page offset?

- 12
- 13
- 16
- Unknown

A: 16

2^{16} is 64k, so we use the lowest 16 bits as the page offset.

Example translation: 64kB pages



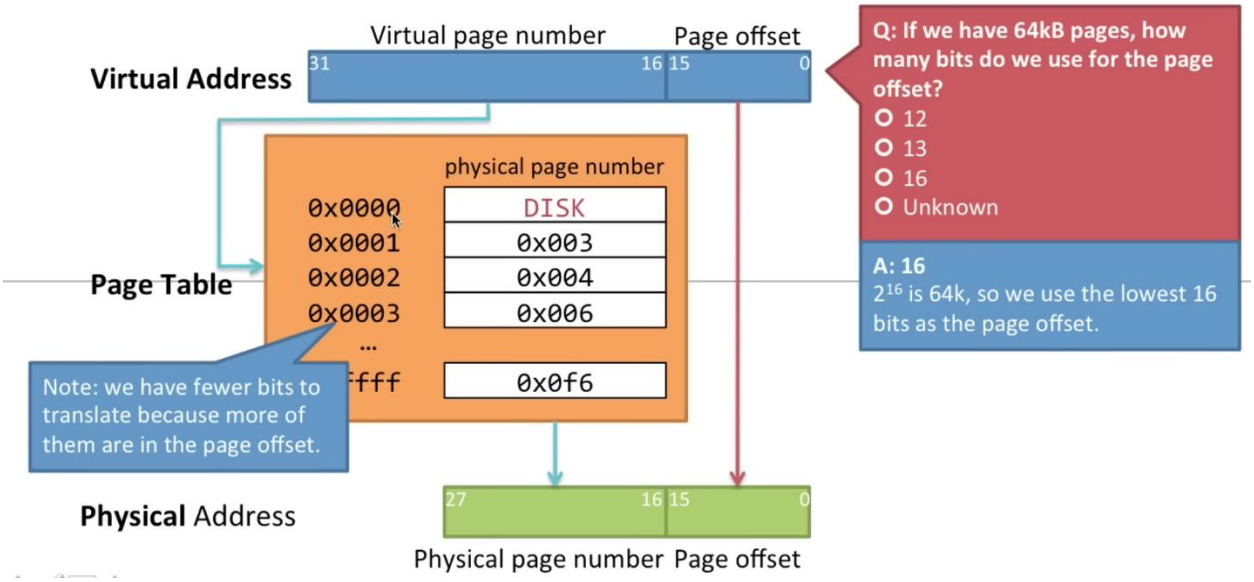
Q: If we have 64kB pages, how many bits do we use for the page offset?

- 12
- 13
- 16
- Unknown

A: 16

2^{16} is 64k, so we use the lowest 16 bits as the page offset.

Example translation: 64kB pages

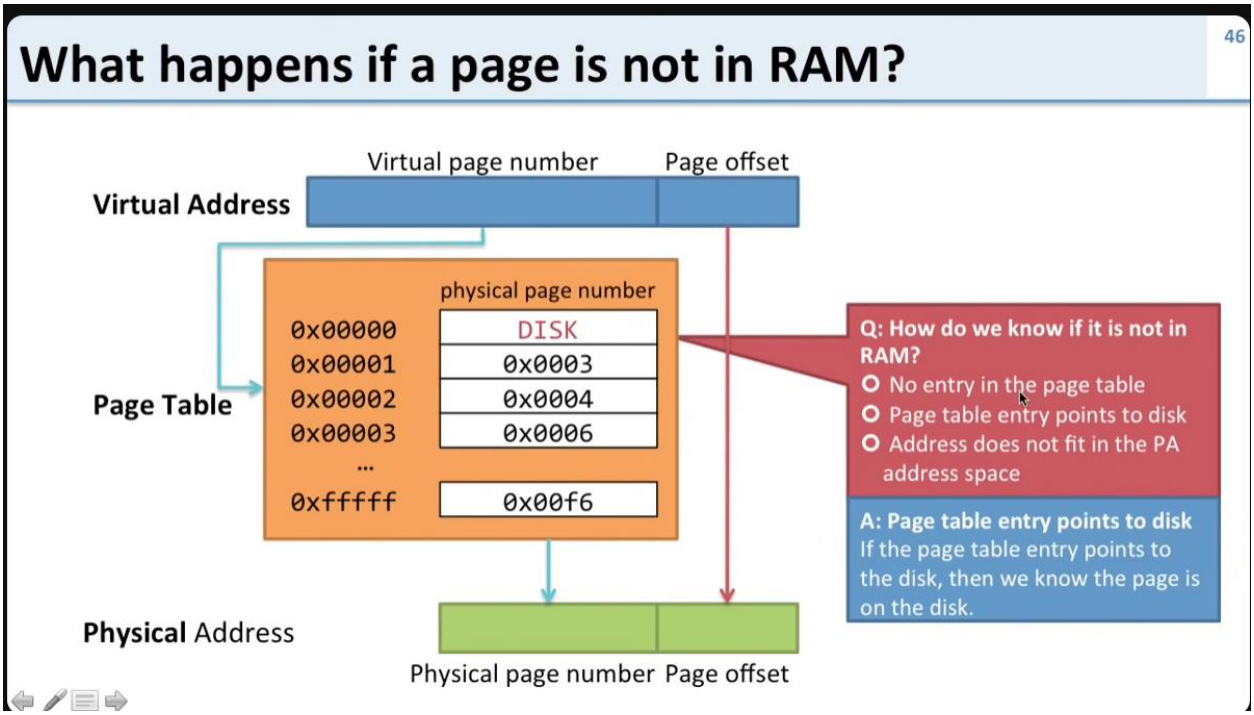


Virtual Memory: 8 Page Faults

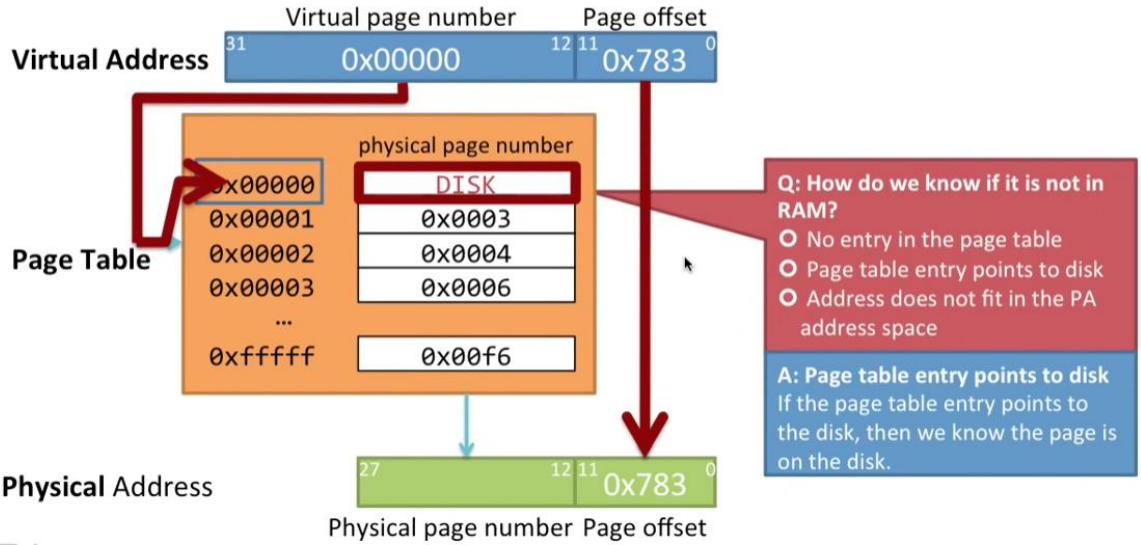
https://www.youtube.com/watch?v=bShqyf-hDfg&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=9

Page faults

What happens when the data is not in RAM?



What happens if a page is not in RAM?



Q: How do we know if it is not in RAM?

- No entry in the page table
- Page table entry points to disk
- Address does not fit in the PA address space

A: Page table entry points to disk

If the page table entry points to the disk, then we know the page is on the disk.

What happens if a page is not in RAM?

- Page Table Entry says the page is on disk
- Hardware (CPU) generates a **page fault exception**
- The hardware jumps to the OS page fault handler to clean up
  - The OS chooses a page to evict from RAM and write to disk
  - If the page is **dirty**, it needs to be written back to disk first
  - The OS then reads the page from disk and puts it in RAM
  - The OS then changes the Page Table to map the new page
- The OS jumps back to the instruction that caused the page fault.
  - (This time it won't cause a page fault since the page has been loaded.)

"Dirty" means the data has been changed (written). If the page has not been written since it was loaded from disk, then it doesn't have to be written back.

Q: How long does this take?

- No time
- A short time
- A long time
- An amazingly, incredibly, painfully long time

A: An amazingly, incredibly, painfully long time

Disks are *much* slower than RAM, so every time you have a page fault it takes an amazingly, incredibly, painfully long time.

How long does a page fault take?

- Page Table Entry says the page is on disk ~1 cycles
- Hardware (CPU) generates a **page fault exception** ~100 cycles
- The hardware jumps to the OS page fault handler to clean up ~10,000 cycles
  - The OS chooses a page to evict from RAM and write to disk
  - If the page is **dirty**, it needs to be written back to disk first ~40,000,000 cycles
  - The OS then reads the page from disk and puts it in RAM ~40,000,000 cycles
  - The OS then changes the Page Table to map the new page ~1,000 cycles
- The OS jumps back to the instruction that caused the page fault.
  - (This time it won't cause a page fault since the page has been loaded.) ~10,000 cycles

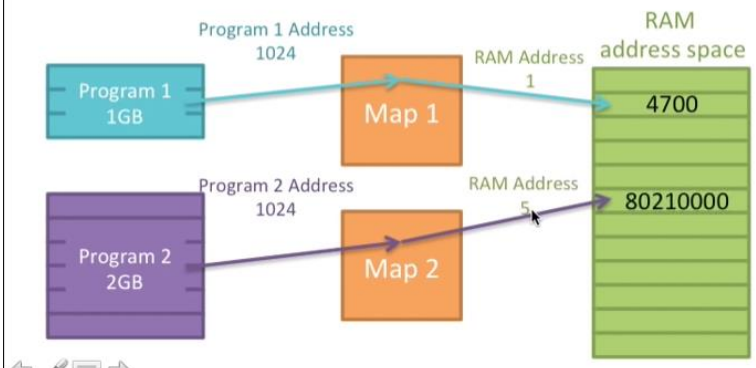
In the time it takes to do handle one page fault
you could execute 80 million cycles on a modern CPU.

Virtual Memory: 9 Memory Protection

https://www.youtube.com/watch?v=uDzXXnNy544&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=9

Using virtual memory to protect applications

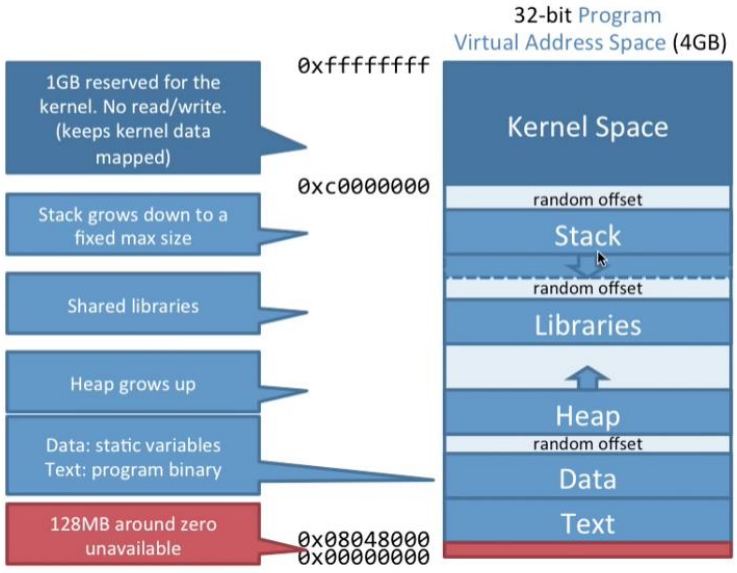
- If each program has its own Page Table, then we can map each program's virtual addresses to unique physical addresses
- Prevents programs from accessing each other's data



Program address space in Linux

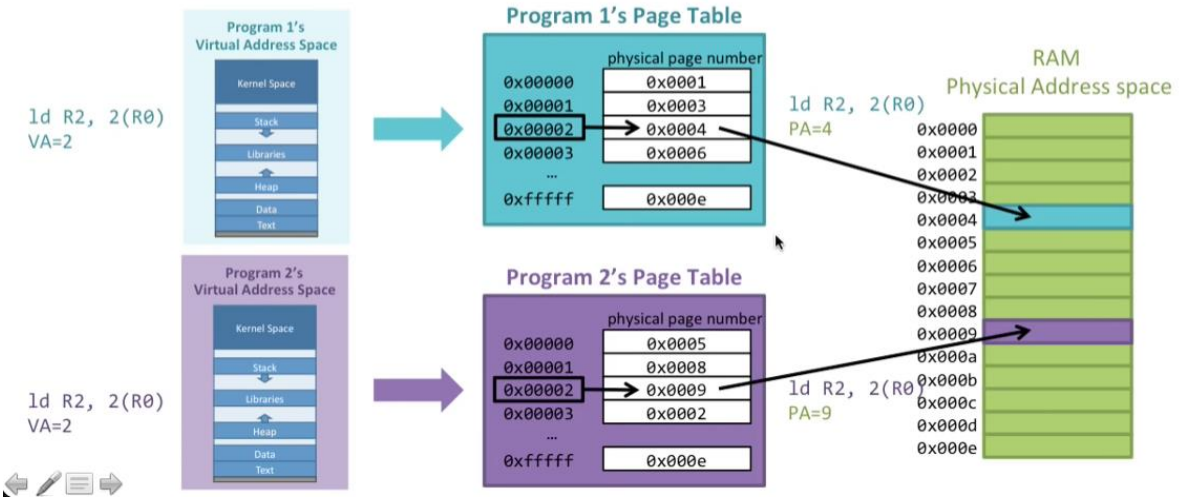
58

- Each program has it's own 32-bit virtual address space
- Linux defines how that address space is used:
  - 1GB reserved for kernel
  - Program static data at the bottom
  - Heap grows up
  - Stack grows down (and has a fixed maximum size)
- Random offsets to enhance security
  - (You never know exactly where a certain piece of data/code will be.)



How do we provide separate mappings?

- Each process needs its own Page Table
- OS makes sure they only map to the same physical address when we want sharing



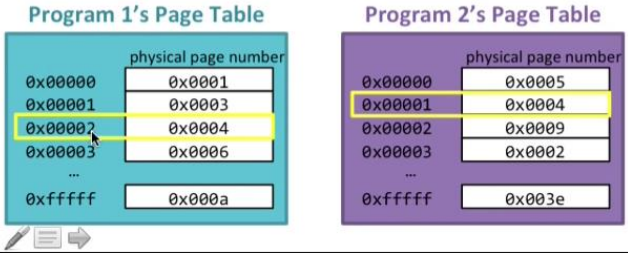
Question: Memory protection

Q: What addresses (pages) could these two programs write to that could cause them to corrupt each other?

- Program 1 0x0001 and Program 2 0x0001
- Program 1 0x0002 and Program 2 0x0002
- Program 1 0x0002 and Program 2 0x0001
- Program 1 0xffff and Program 2 0xffff

A: Program 1 0x0002 and Program 2 0x0001

Program 1's page table maps VA 0x0002 to PA 0x0004.
Program 2's page table maps VA 0x0001 to PA 0x0004.
If both programs write to these addresses they will change the same physical RAM location.



Virtual Memory: 10 Making Virtual Memory Fast

https://www.youtube.com/watch?v=uyrSn3qbZ8U&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=10

Question: How much does virtual memory cost?

Q: What do we have to do for each memory access with virtual memory?

- Load data from disk
- Access the page table in RAM
- Translate the address
- Update the cache
- Have the operating system update the page table
- Access the data in RAM

A:
1. Access the page table in RAM
2. Translate the address
3. Access the data in RAM

This is a lot of work for every memory access…and remember we have an average of 1.33 accesses for each instruction!

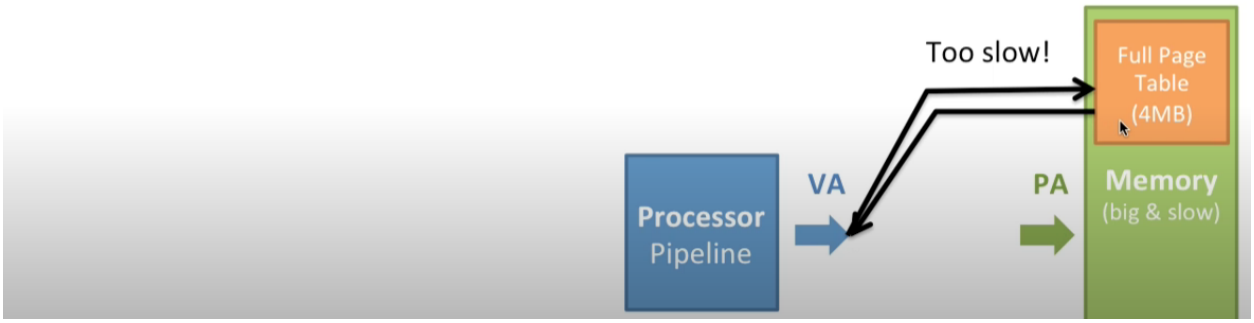
How do we do virtual memory in practice?

- VM is great:
  - Unlimited programs/memory, protection, flexibility, etc.
- But it comes at a high cost:
  - ***Every*** memory operation has to look up in the page table
  - Need to access **1) the page table** and **2) the memory address** (2x memory accesses)

  (Remember, 1.33 memory accesses per instruction. This is going to hurt.)
- How can we make a page table look up ***really really*** fast?
  - Software would be far too slow
    (e.g., an extra 5 instructions for every memory access would kill performance)
- Perhaps a hardware page table **cache**?

Making VM fast: the TLB

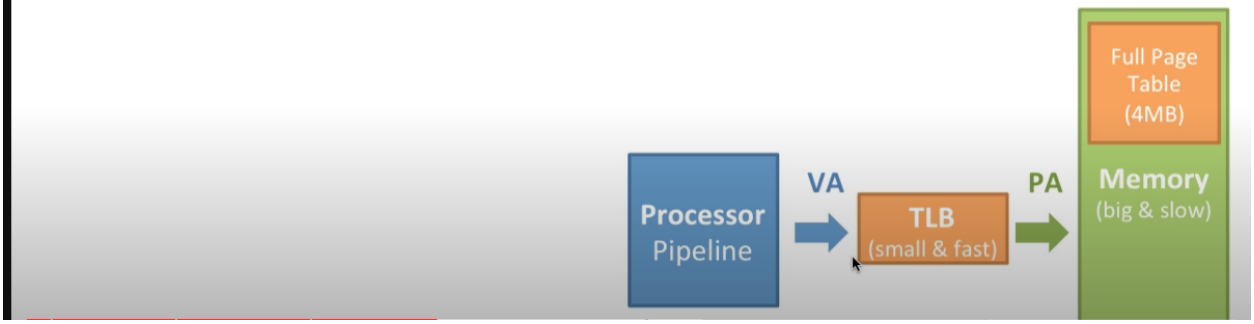
- To make VM fast we add a special **Page Table cache:** the **Translation Lookaside Buffer (TLB)**



Making VM fast: the TLB

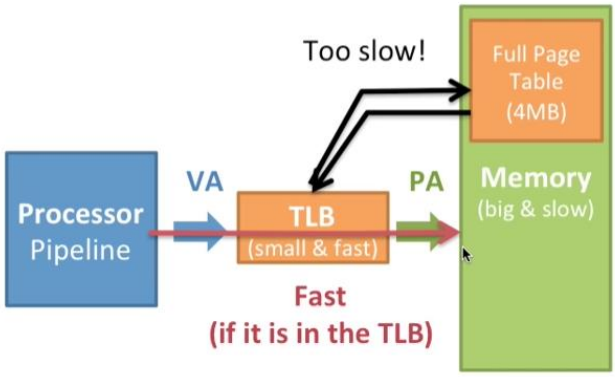
69

- To make VM fast we add a special **Page Table cache:** the **Translation Lookaside Buffer (TLB)**
  - Fast: less than 1 cycle (have to do it for every memory access)
  - Very similar to a cache



Making VM fast: the TLB

- To make VM fast we add a special **Page Table cache:** the **Translation Lookaside Buffer (TLB)**
  - Fast: less than 1 cycle (have to do it for every memory access)
  - Very similar to a cache



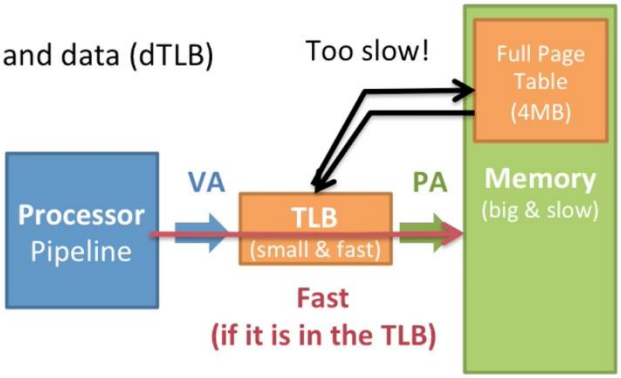
Making VM fast: the TLB

- To make VM fast we add a special **Page Table cache:** the **Translation Lookaside Buffer (TLB)**
  - Fast: less than 1 cycle (have to do it for every memory access)
  - Very similar to a cache

- To be fast, TLBs must be small:
  - Separate TLBs for instructions (iTLB) and data (dTLB)
  - 64 entries, 4-way (4kB pages)
  - 32 entries, 4-way (2MB pages)

  (Page Table is 1M entries)

Lots of locality!
Miss rates are typically only a few percent.



What can happen when we access memory?

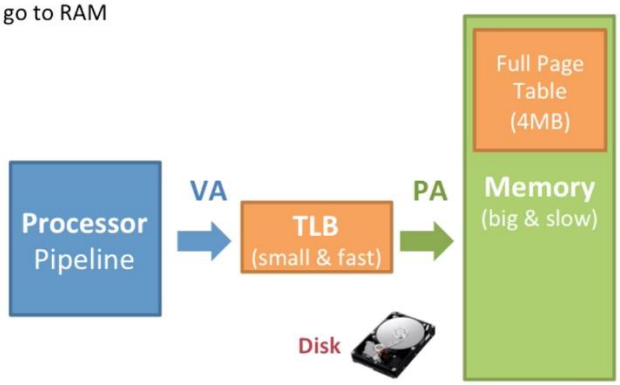
Good: Page in RAM

- **PTE in the TLB**
  - **Excellent**
  - **<1** cycle to translate, then go to RAM (or cache)
- **PTE not in the TLB**
  - **Poor**
  - **20-1000** cycles to load PTE from RAM, then go to RAM

With 1.33 memory accesses per instruction we can't afford 20-1000 cycles very often.

Bad: Page not in RAM

- **PTE in the TLB** (unlikely)
  - **Horrible**
  - 1 cycle to know it's on disk
  - **~80M** cycles to get it from disk
- **PTE not in the TLB**
  - **(ever so slightly more) horrible**
  - 20-1000 cycles to know it's on disk
  - **~80M** cycles to get it from disk



Question: Making TLBs (seem) bigger

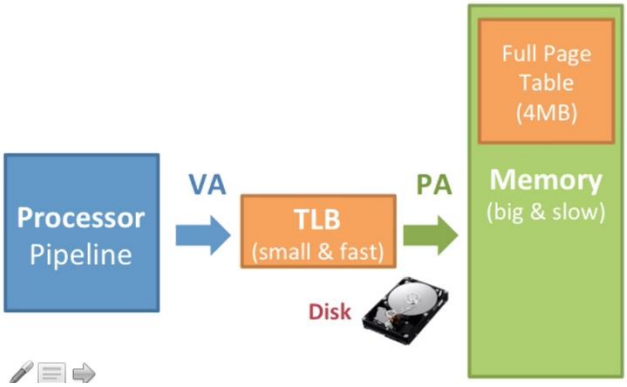
Q: TLBs are small. How can we make them effectively bigger without slowing them down?

- Just make them hold more PTEs!
- Make pages larger
- Add a second TLB that is larger, but a bit slower
- Have hardware to fill the TLB automatically if there is a miss. (E.g., instead of having the OS do it in software.)

A:

1. Make pages larger

This increases the reach of the TLB because you need fewer pages to cover more data.



64 4kB pages = 256kB of data

Question: Making TLBs (seem) bigger

Q: TLBs are small. How can we make them effectively bigger without slowing them down?

- Just make them hold more PTEs!
- Make pages larger
- Add a second TLB that is larger, but a bit slower
- Have hardware to fill the TLB automatically if there is a miss. (E.g., instead of having the OS do it in software.)

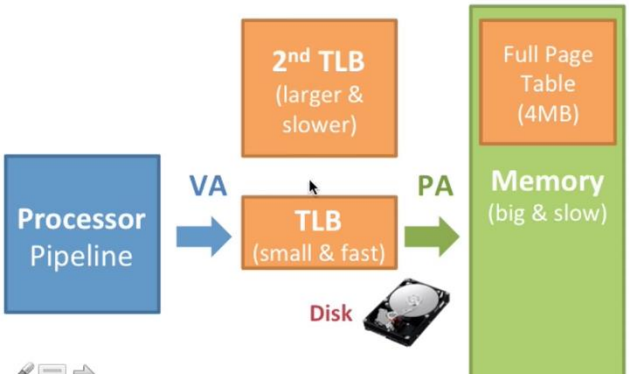
A:

1. Make pages larger

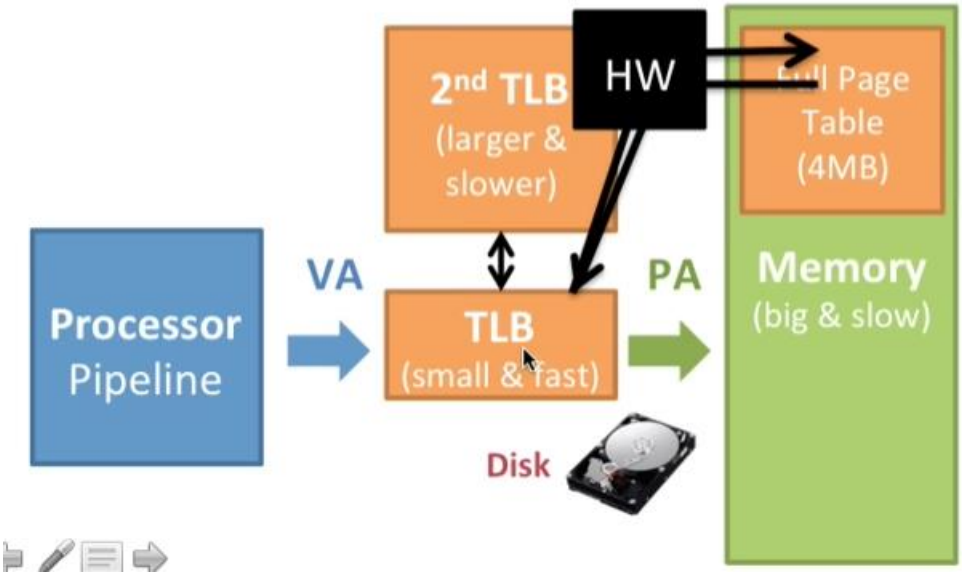
This increases the reach of the TLB because you need fewer pages to cover more data.

2. Add a second TLB that is larger, but a bit slower

Sure. Most processors have a level 2 TLB that is about 8x larger than the level 1 TLB, but also about twice as slow.

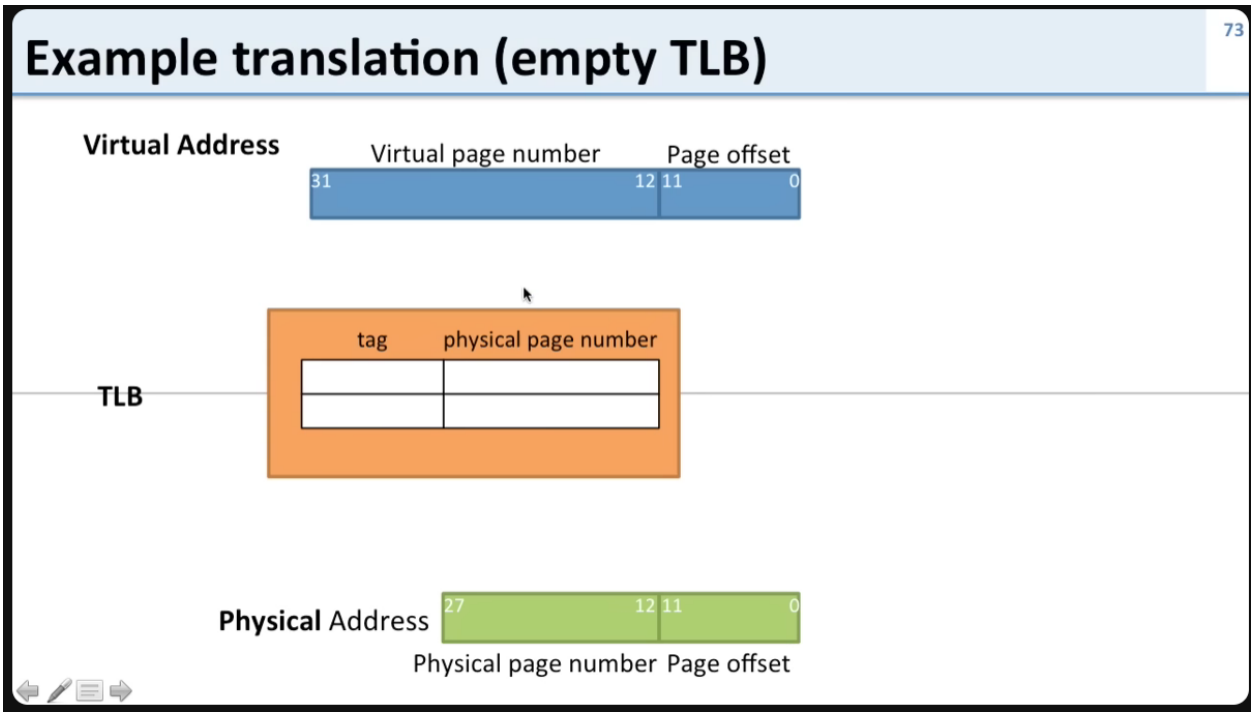


64 4kB pages = 256kB of data
32 2MB pages = 64MB of data

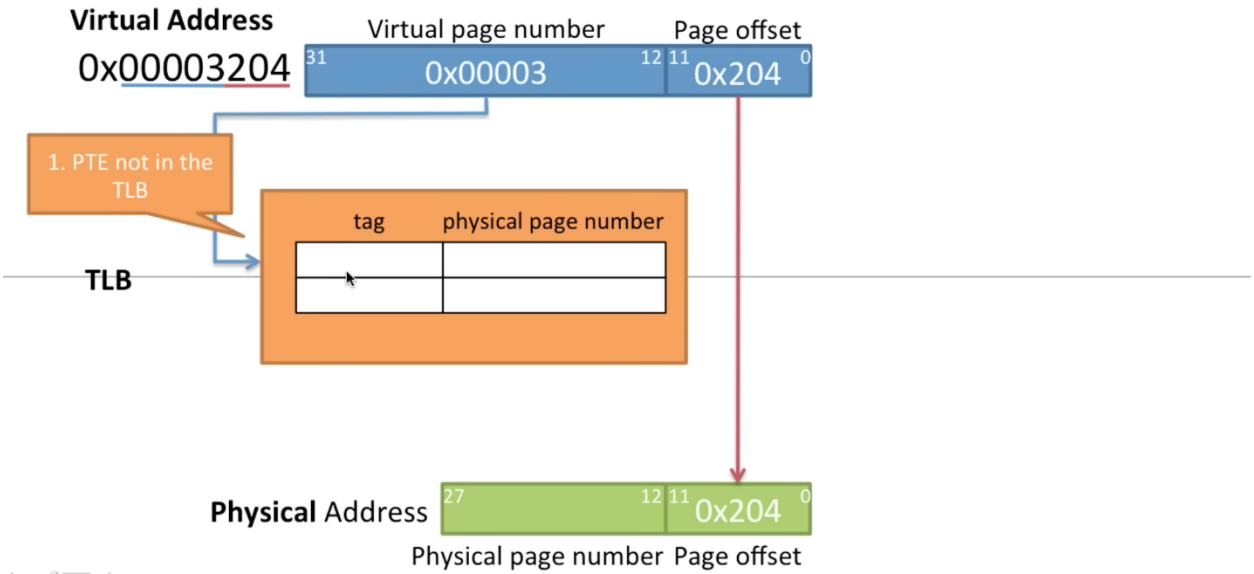


Virtual Memory: 11 TLB Example

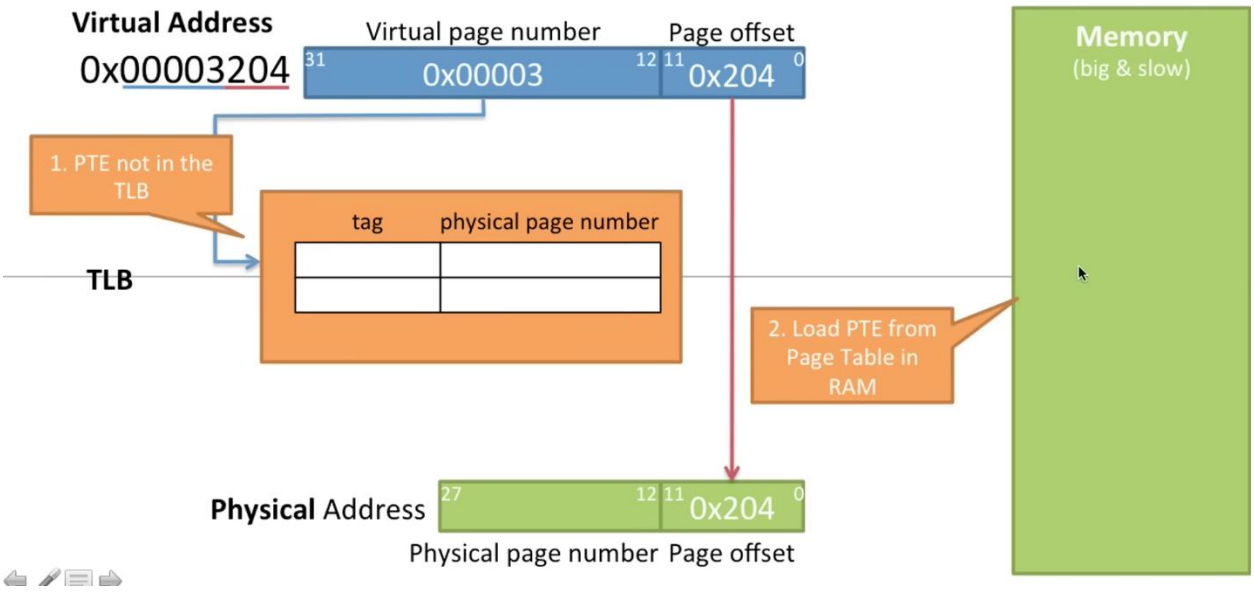
https://www.youtube.com/watch?v=95QpHJX55bM&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=13



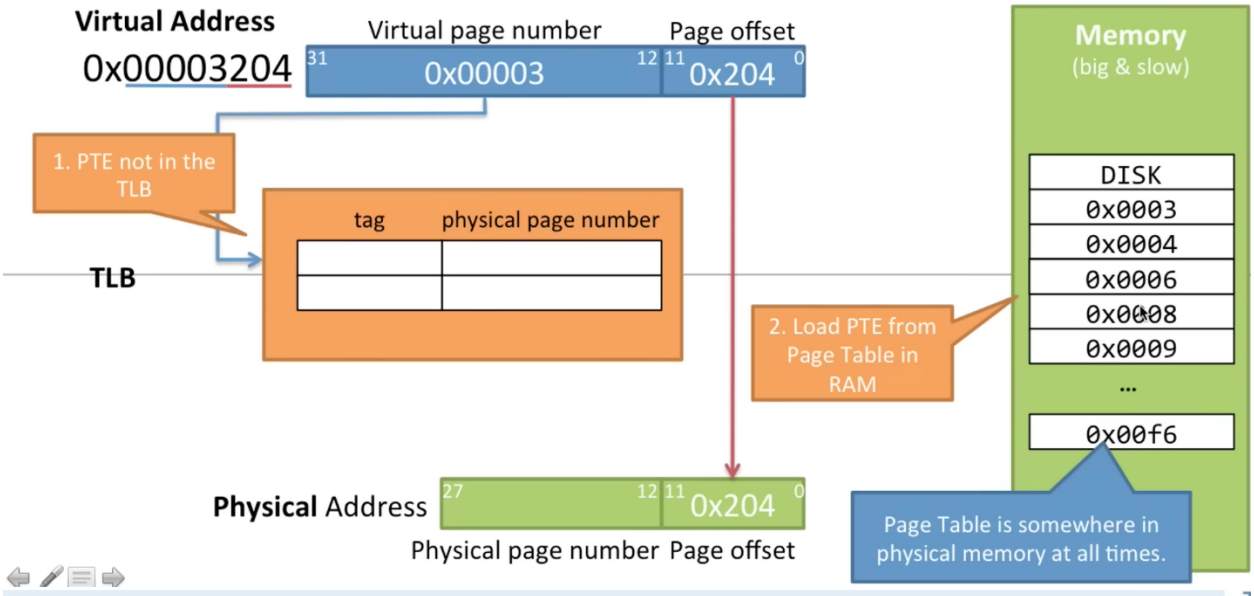
Example translation (empty TLB)



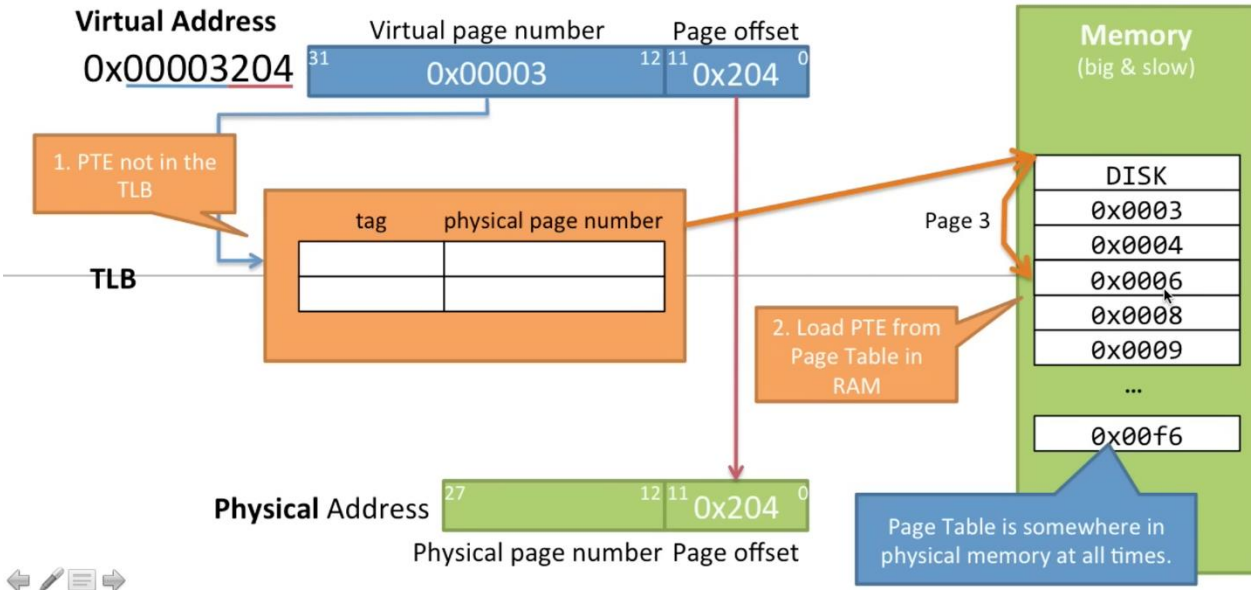
Example translation (empty TLB)



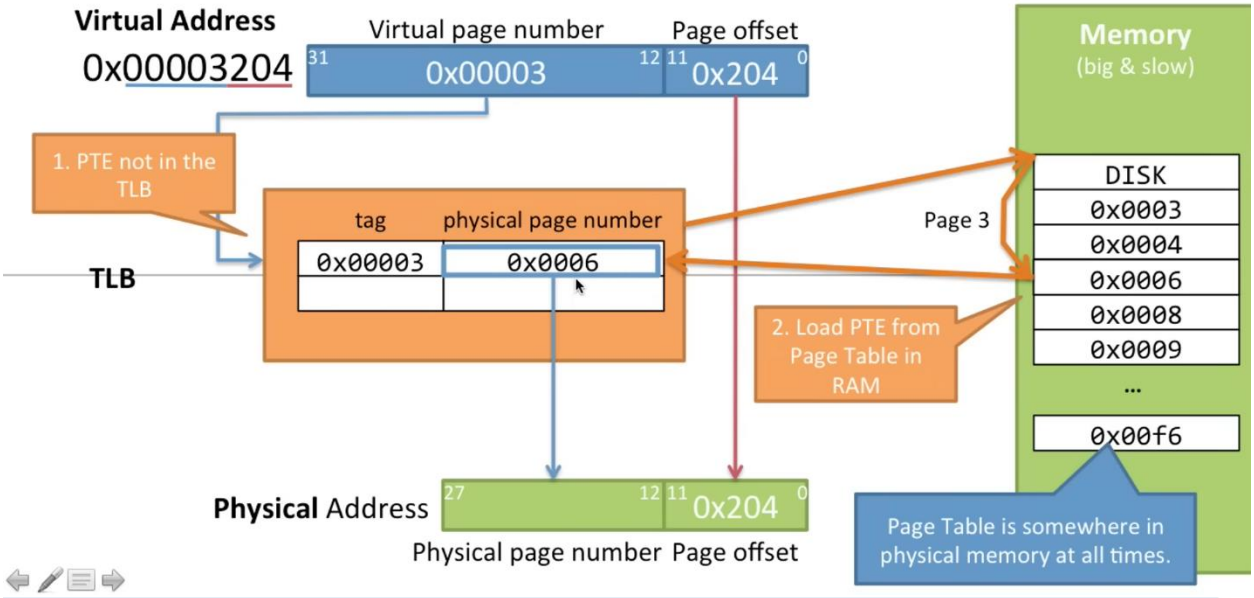
Example translation (empty TLB)



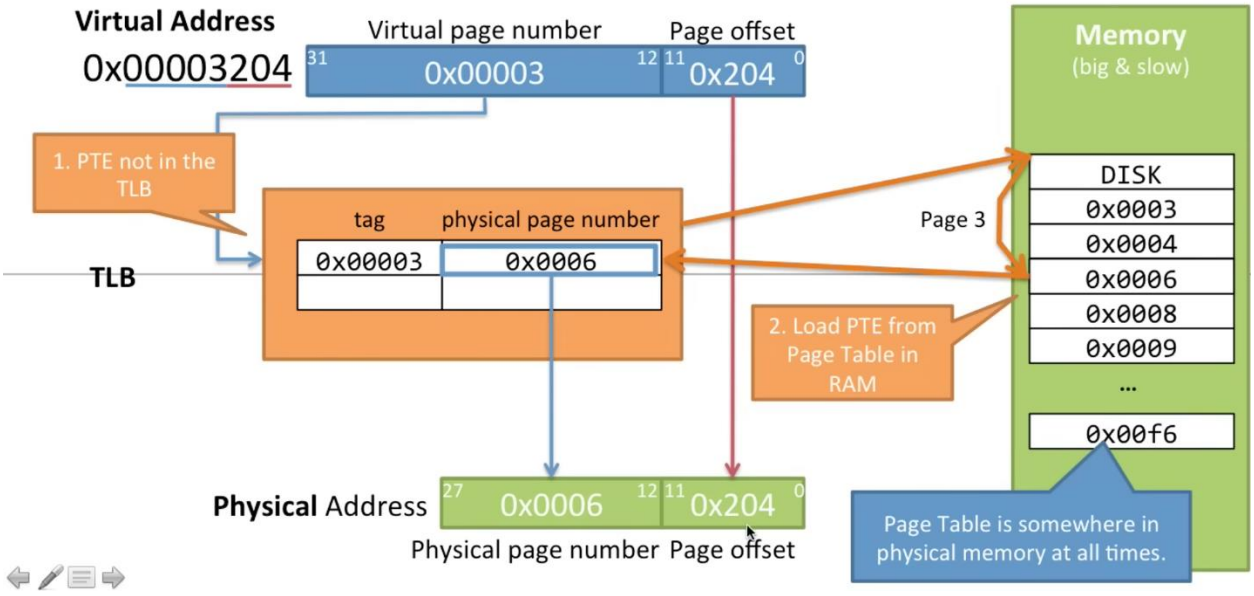
Example translation (empty TLB)



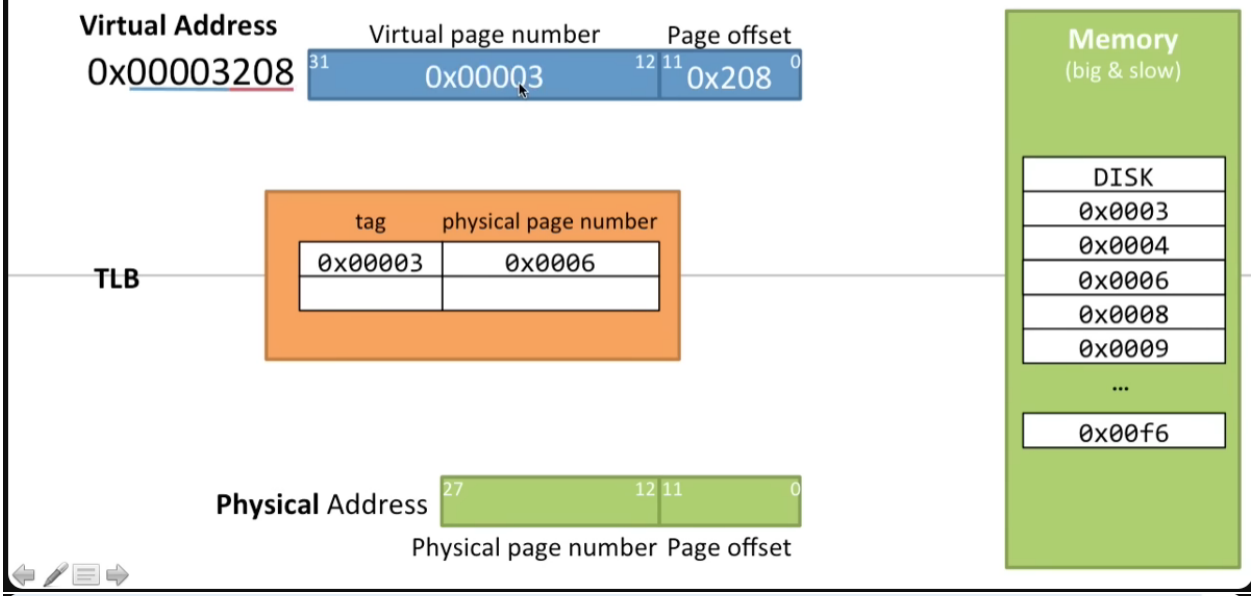
Example translation (empty TLB)



Example translation (empty TLB)

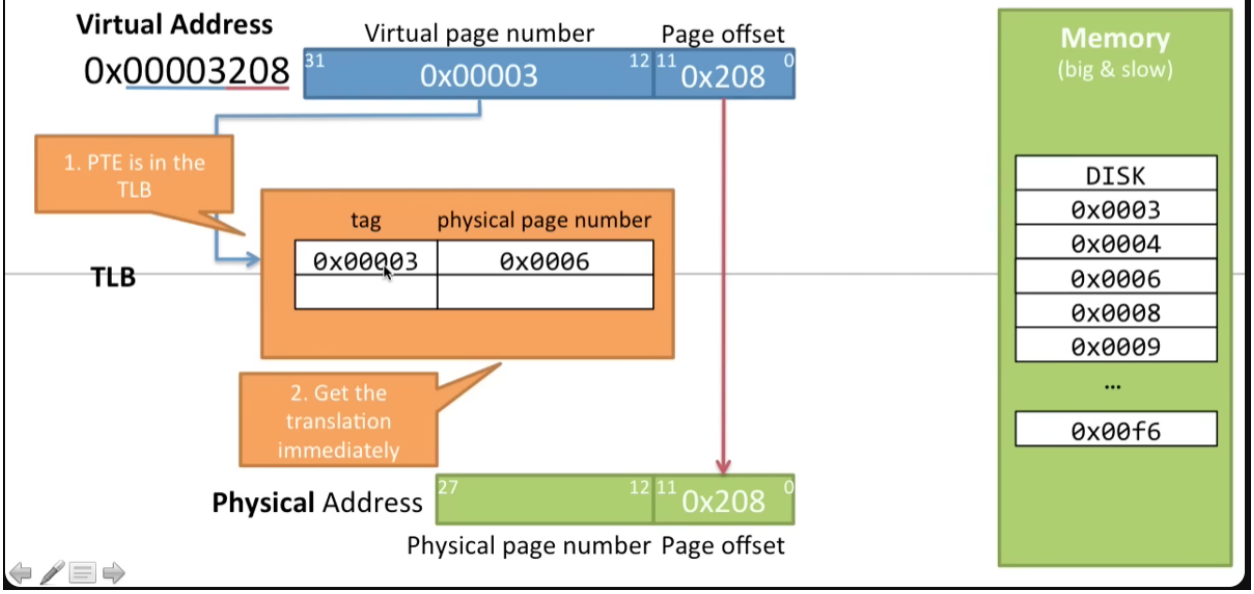


Example translation (TBL hit)

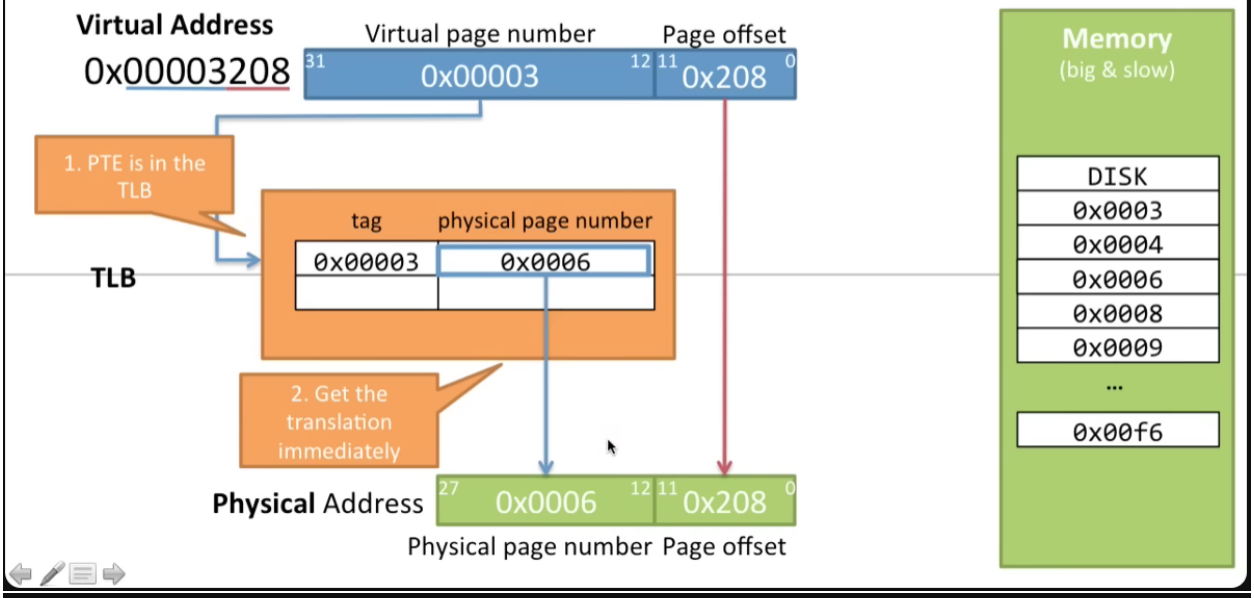


Example translation (TBL hit)

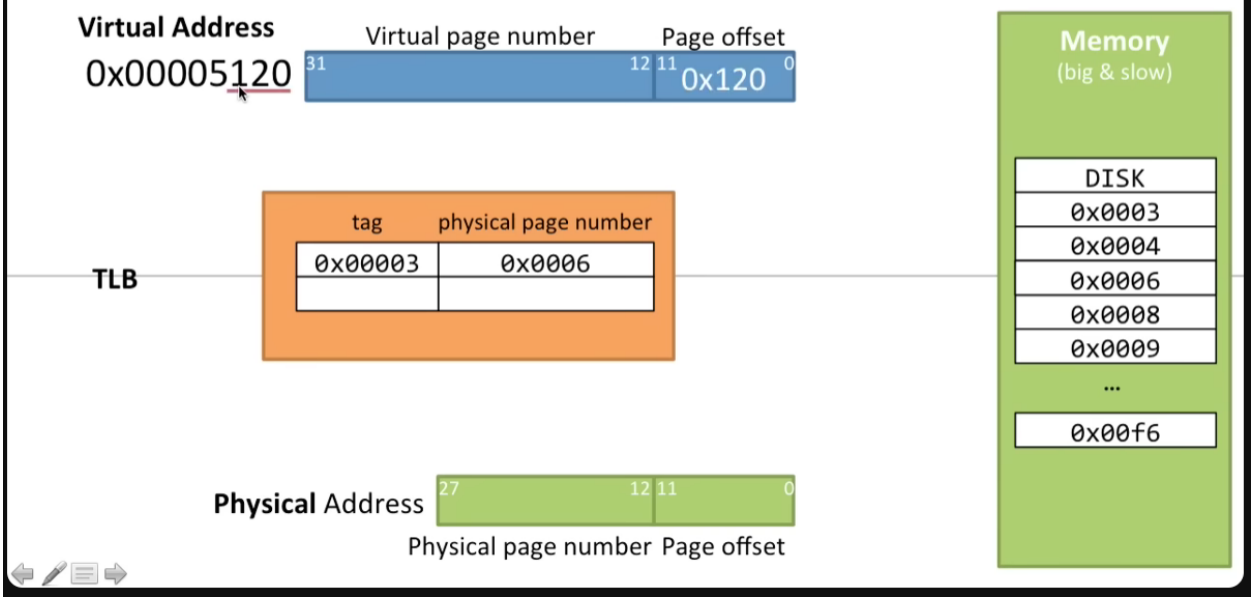
74



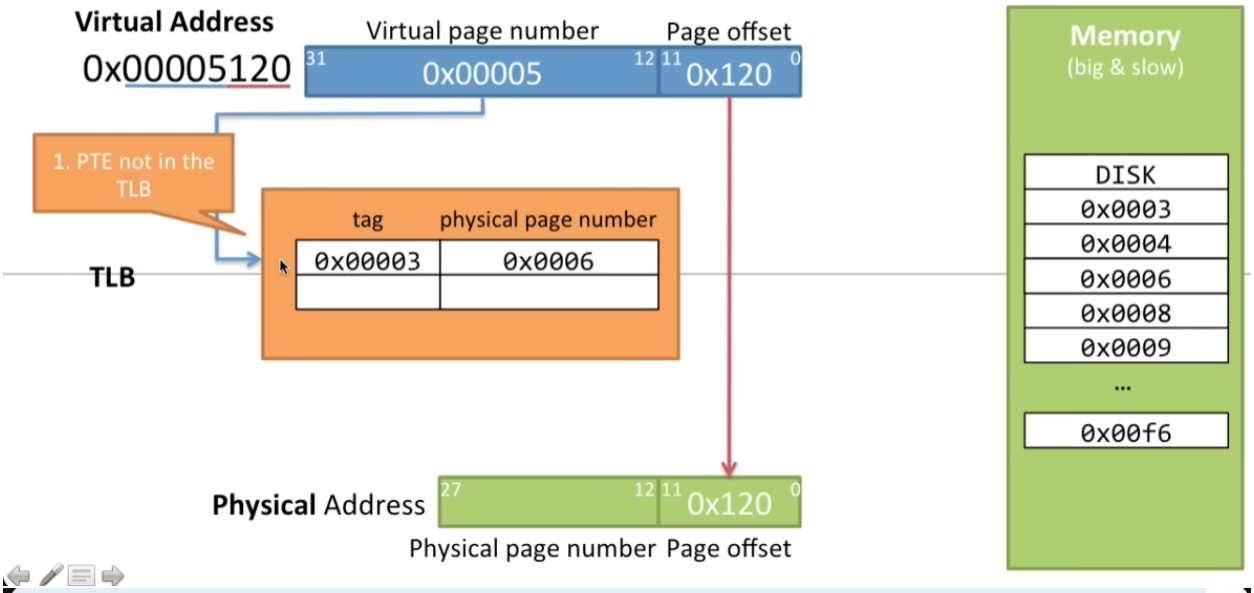
Example translation (TBL hit)



Example translation (TBL miss)

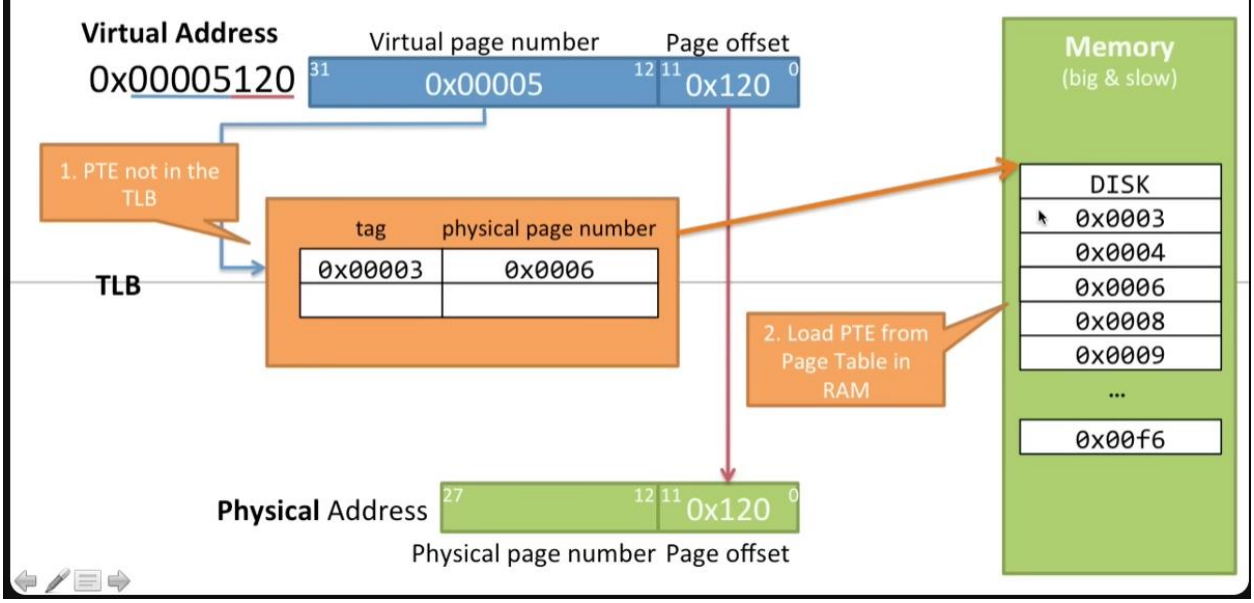


Example translation (TBL miss)

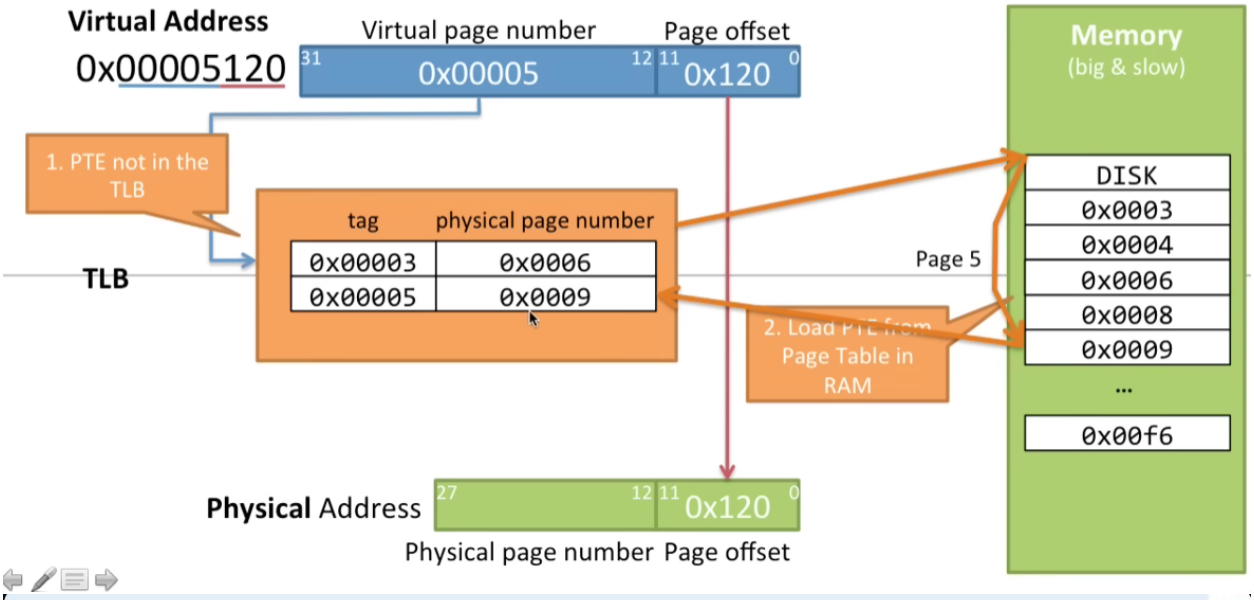


Example translation (TBL miss)

75

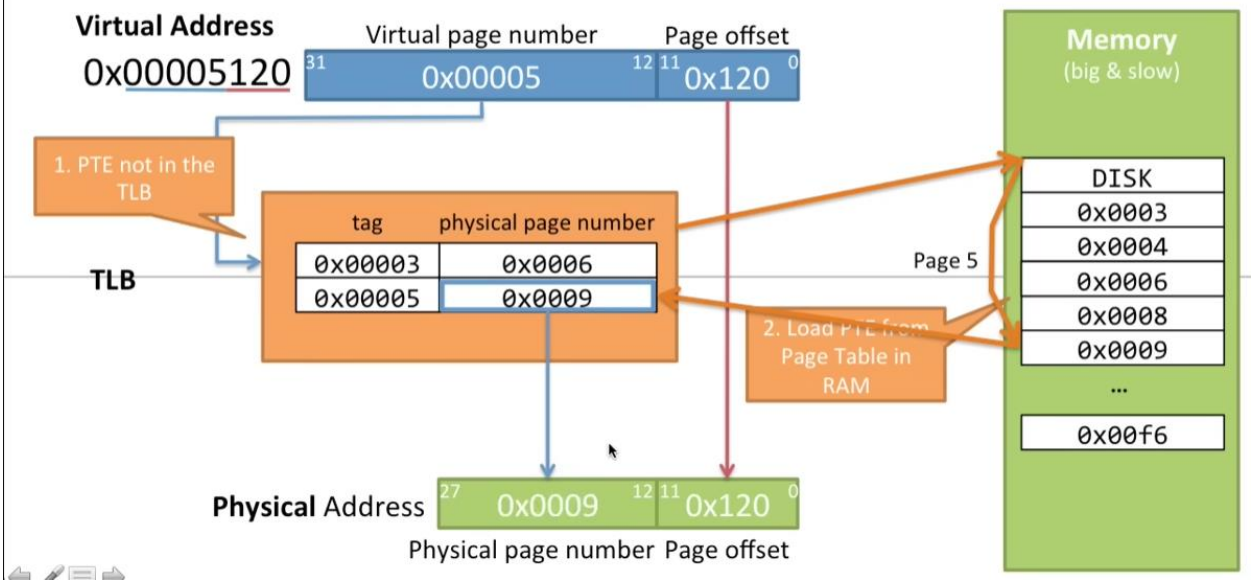


Example translation (TBL miss)

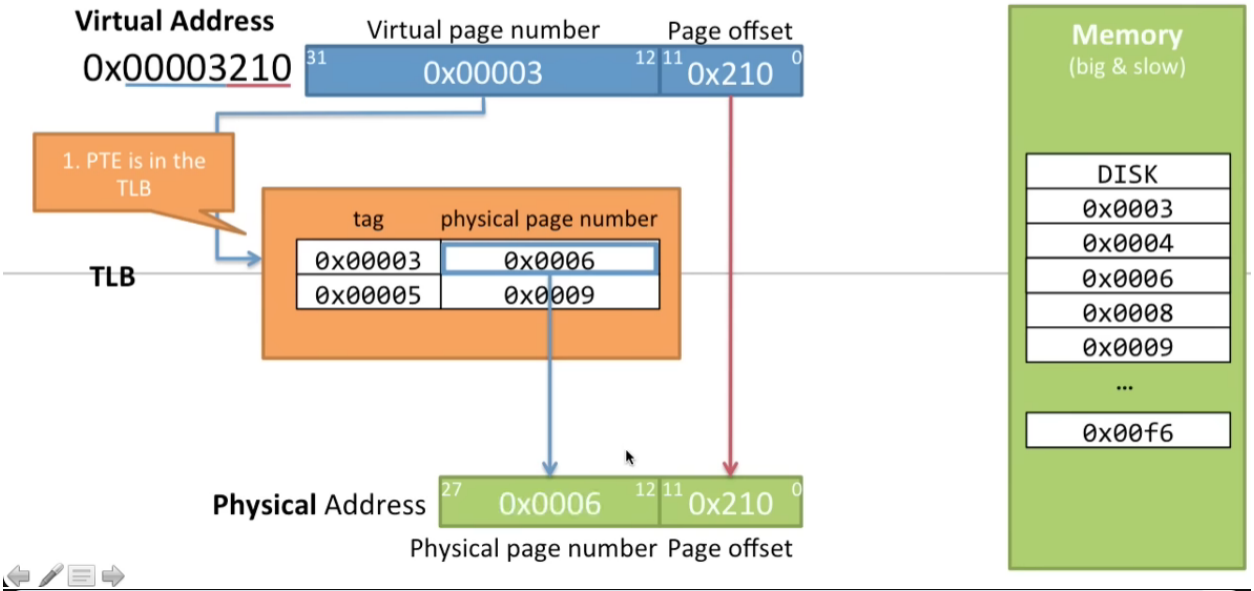


Example translation (TBL miss)

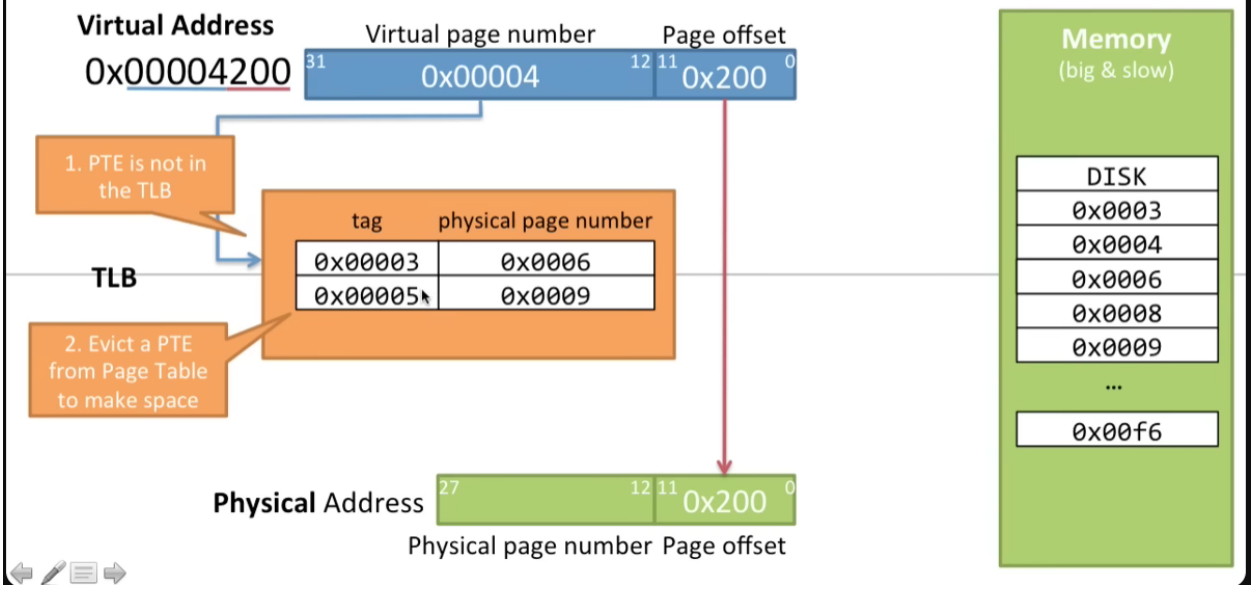
75



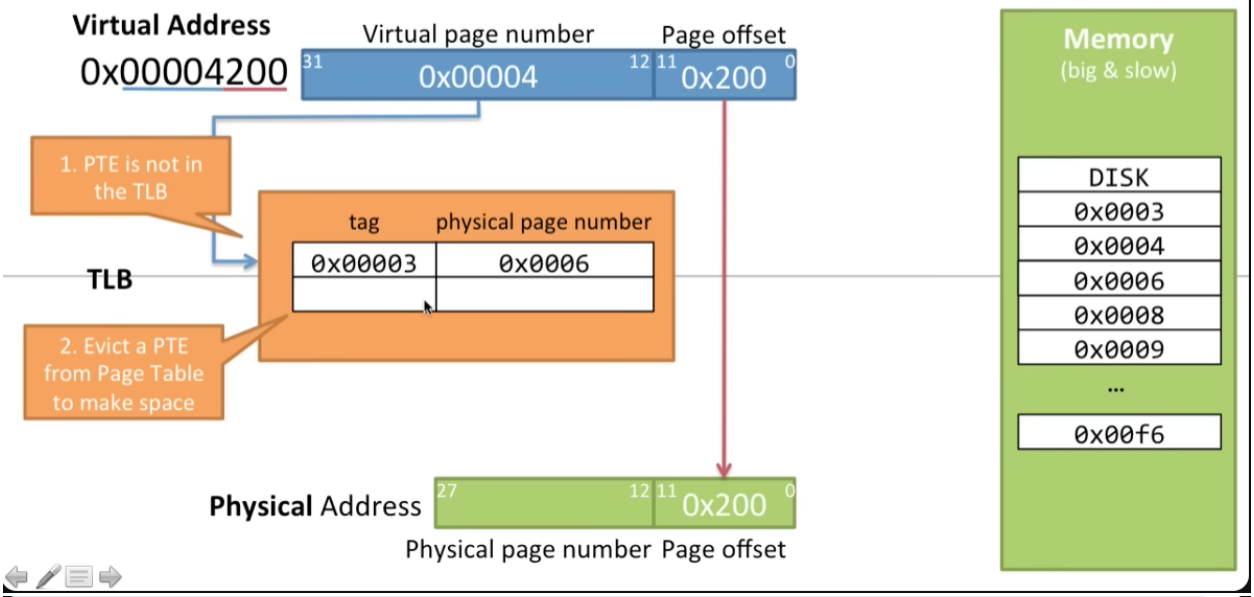
Example translation (TLB hit)



Example translation (TLB miss + eviction)

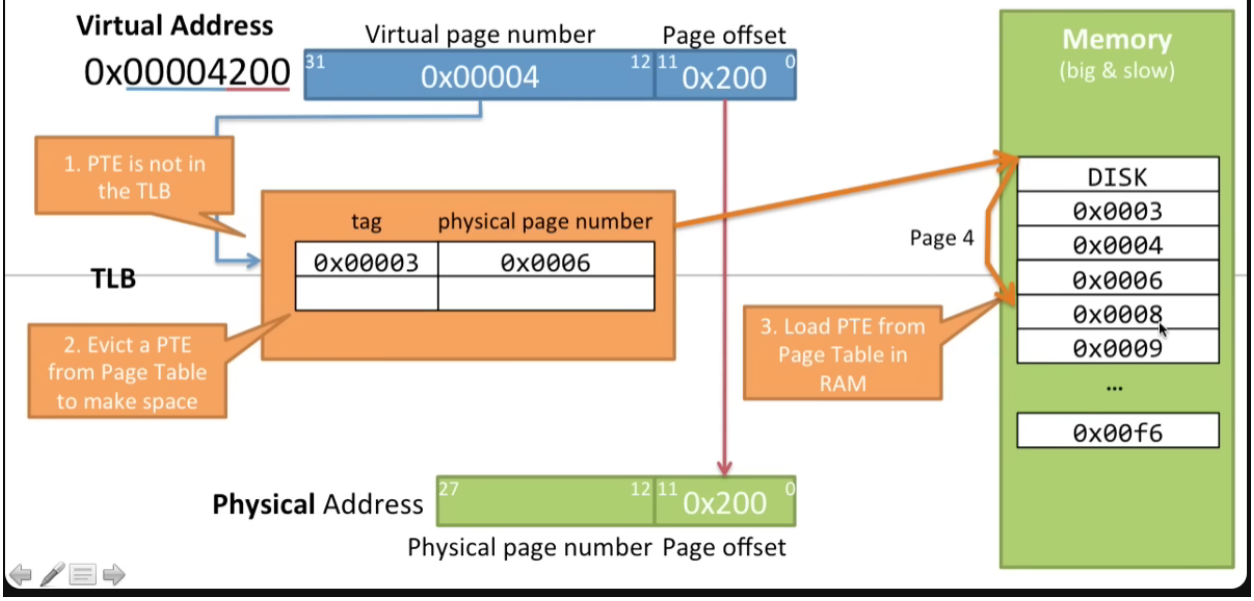


Example translation (TLB miss + eviction)



Example translation (TLB miss + eviction)

78



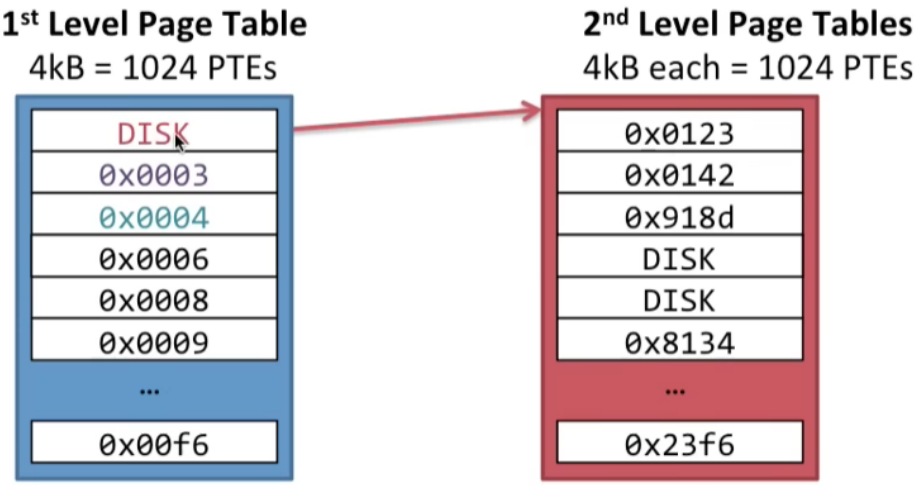
Virtual Memory: 12 Multi-level Page Tables

https://www.youtube.com/watch?v=Z4kSOv49GNc&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=12

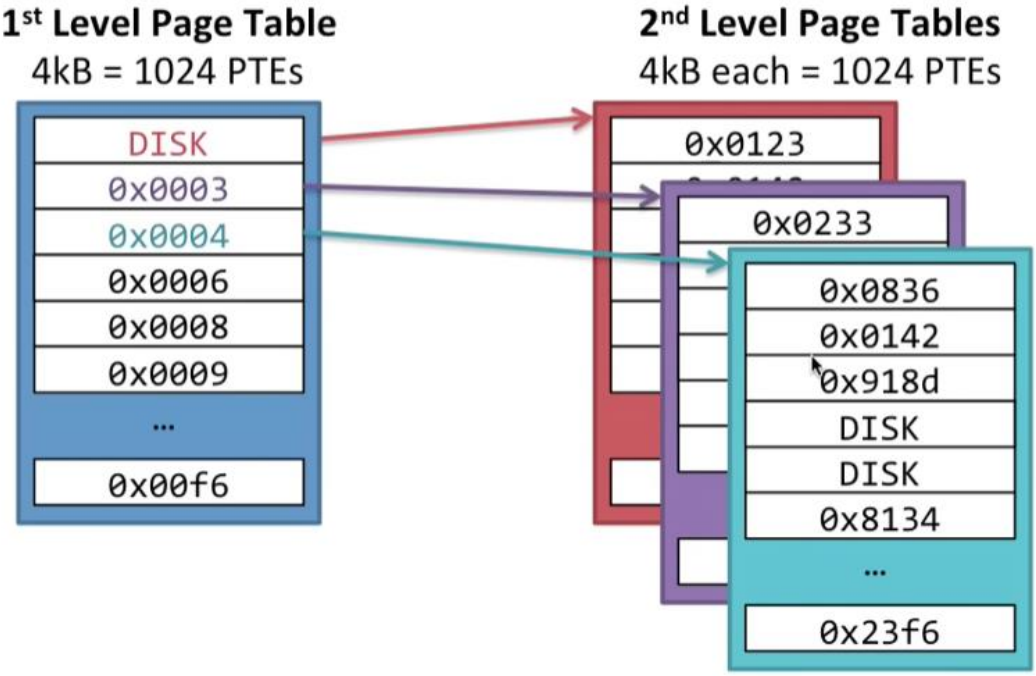
Page table size

- For 32-bit machine with 4kB pages we need:
  - 1M Page Table Entries (32 bits – 12 bits for page offset = 20 bits, 2^{20}=1M)
  - Each PTE is about 4 bytes (20 bits for physical page + permission bits)
  - 4MB total
- Not bad…
  …except **each program needs its own page table**…
  - If we have 100 programs running, we need 400MB of Page Tables!
- And here's the tough part:
  - We **can't swap the page tables out to disk**
  - If the page table is not in RAM, we have no way to access it to find it!
- How can we fix this?

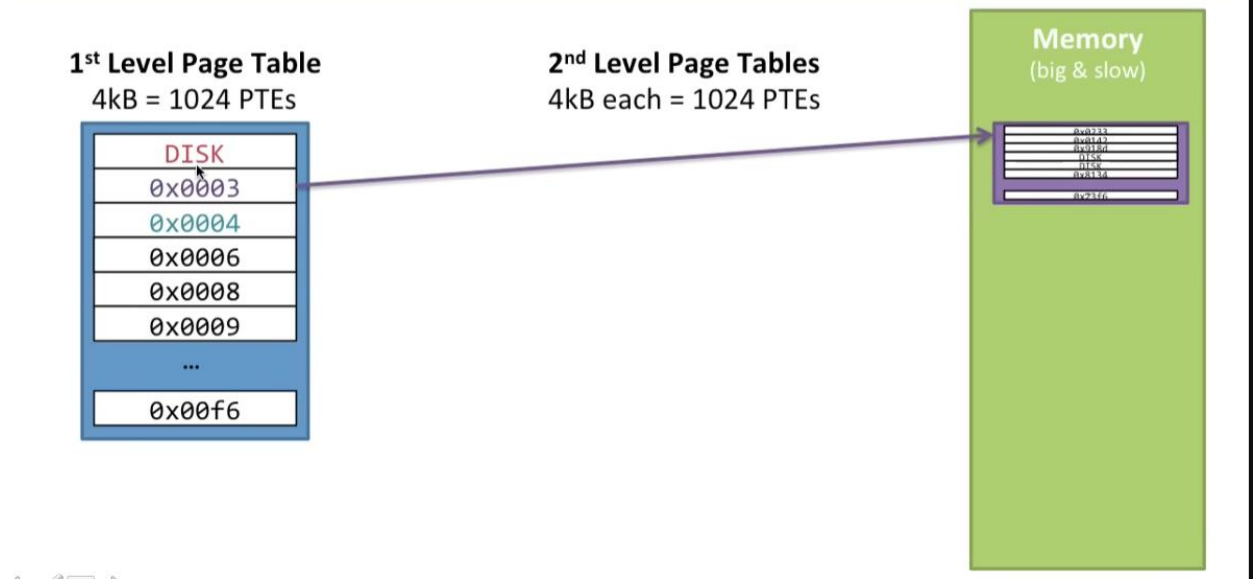
Multi-level page tables



Multi-level page tables

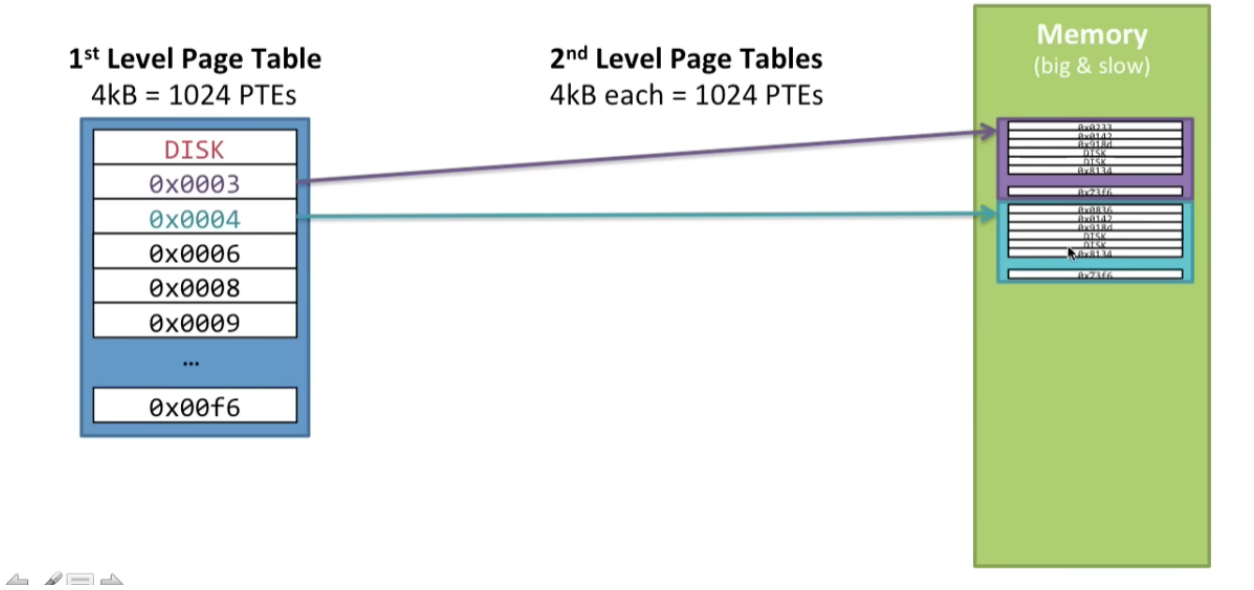


Multi-level page tables

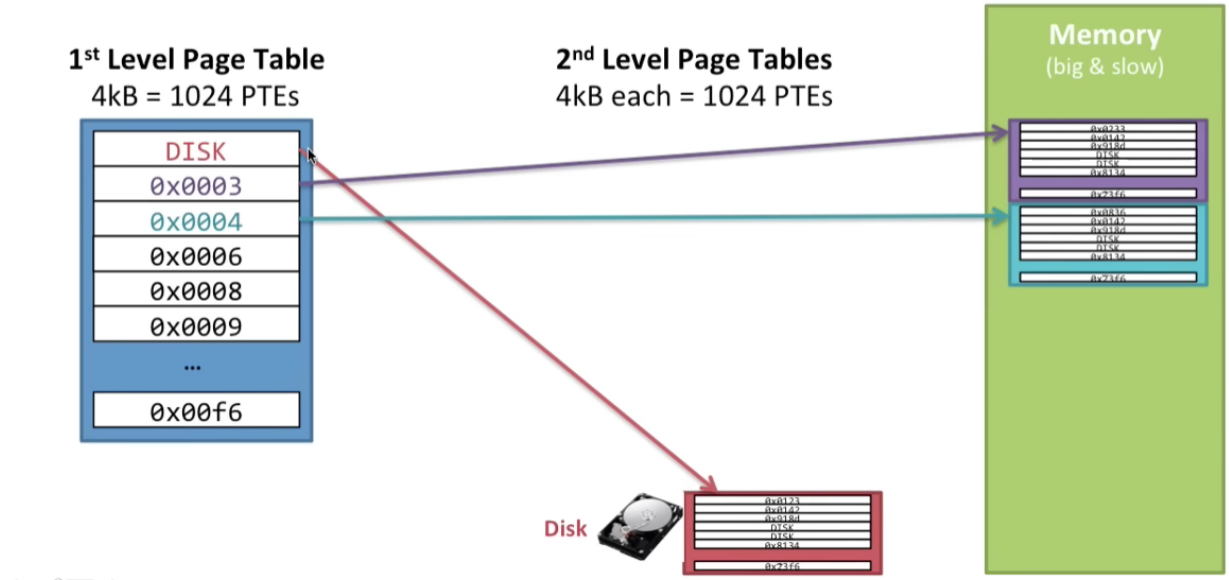


84

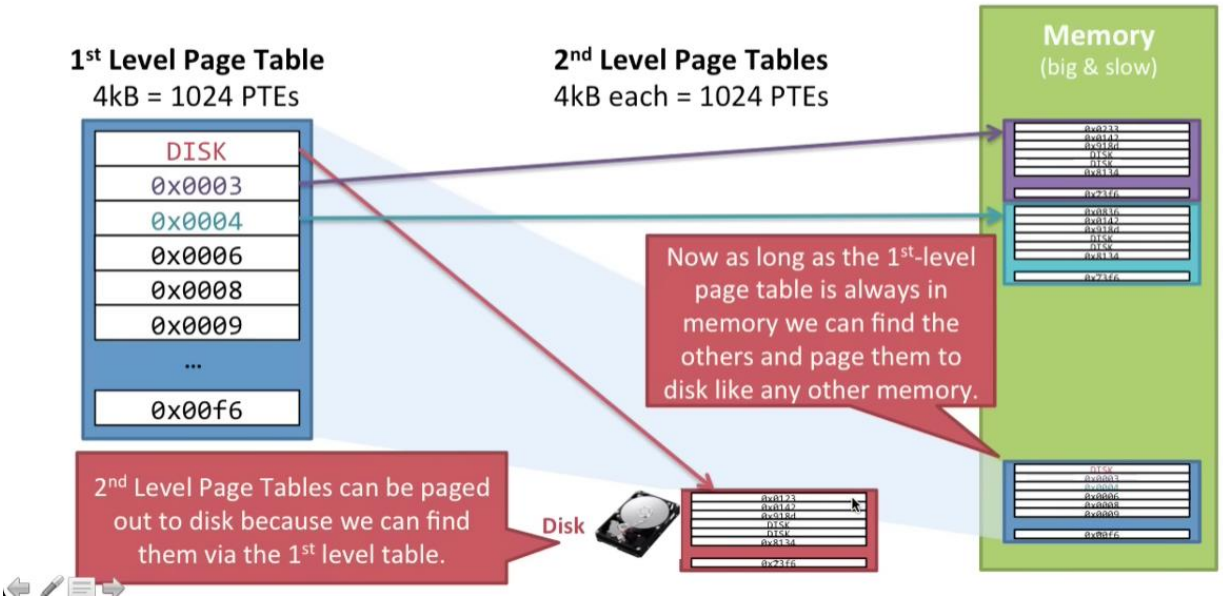
Multi-level page tables



Multi-level page tables



Multi-level page tables



Question: Multi-level page table size

Q: With multi-level page tables, what is the smallest amount of page table data we need to keep in memory for each 32-bit application?

- 4MB+4kB (always need the whole page table plus the 1st-level page)
- 4MB (always need the whole page table in RAM)
- 4kB+4kB (need the 1st-level page table and one 2nd-level page table)
- 4kB (just need the 1st-level page table)

A: 4kB+4kB

We always need the 1st-level page table so we can find the 2nd-level ones.

But, the 1st-level page table only helps us find other page tables. It isn't enough by itself to translate any program addresses. So we need at least one 2nd-level page table to actually translate memory addresses:

One first-level page table: 4kB
One second-level page table: 4kB
4kB+4kB per application (much better than 4MB per application!)

85

Question: Multi-level page table size

Q: With multi-level page tables, what is the smallest amount of page table data we need to keep in memory for each 32-bit application?

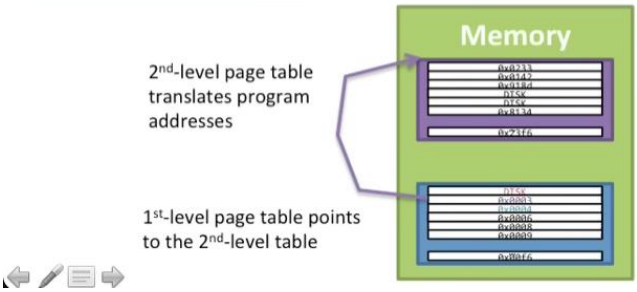
- 4MB+4kB (always need the whole page table plus the 1st-level page)
- 4MB (always need the whole page table in RAM)
- 4kB+4kB (need the 1st-level page table and one 2nd-level page table)
- 4kB (just need the 1st-level page table)

A: 4kB+4kB

We always need the 1st-level page table so we can find the 2nd-level ones.

But, the 1st-level page table only helps us find other page tables. It isn't enough by itself to translate any program addresses. So we need at least one 2nd-level page table to actually translate memory addresses:

One first-level page table: 4kB
One second-level page table: 4kB
4kB+4kB per application (much better than 4MB per application!)



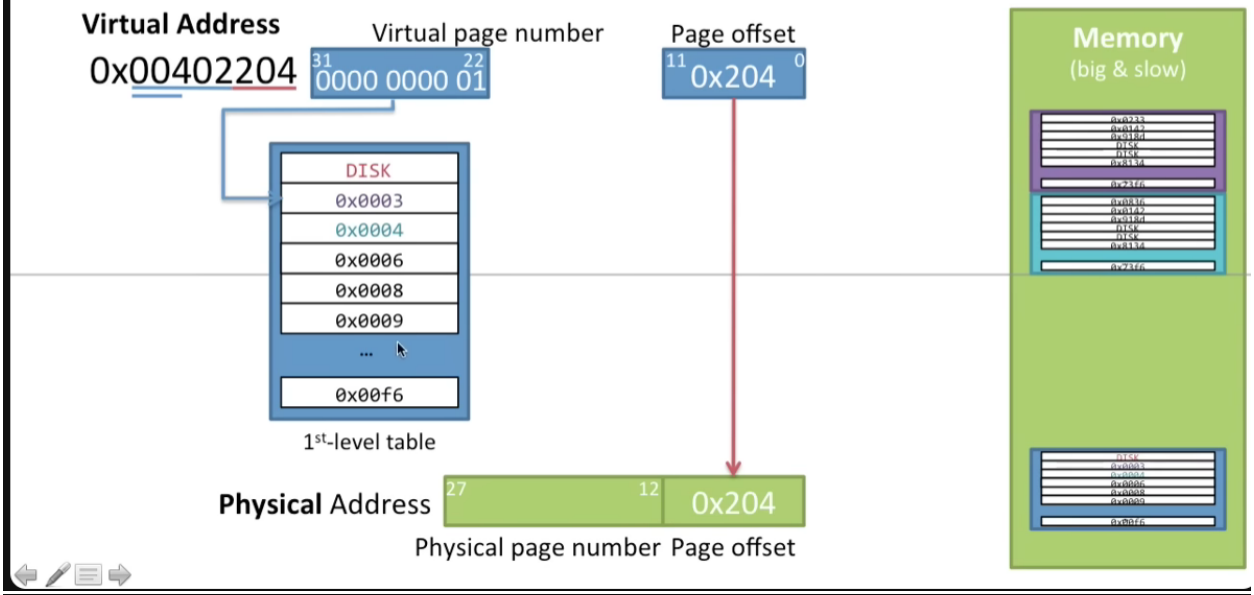
Multi-level page table translation

Virtual Address

0x00402204

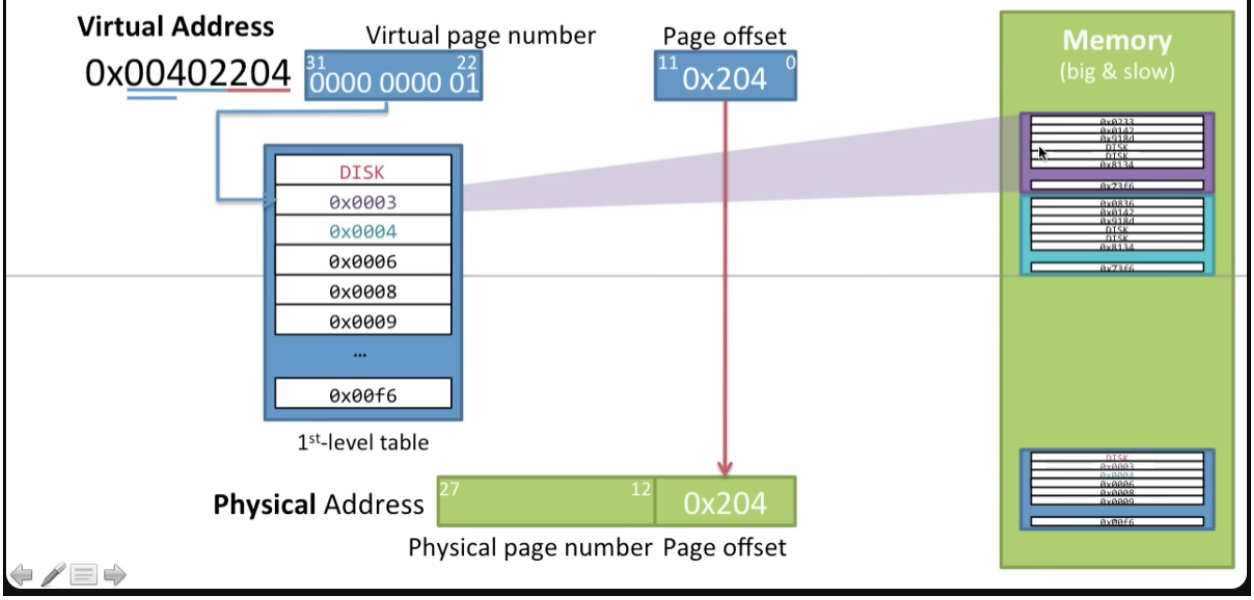
Physical Address

Multi-level page table translation

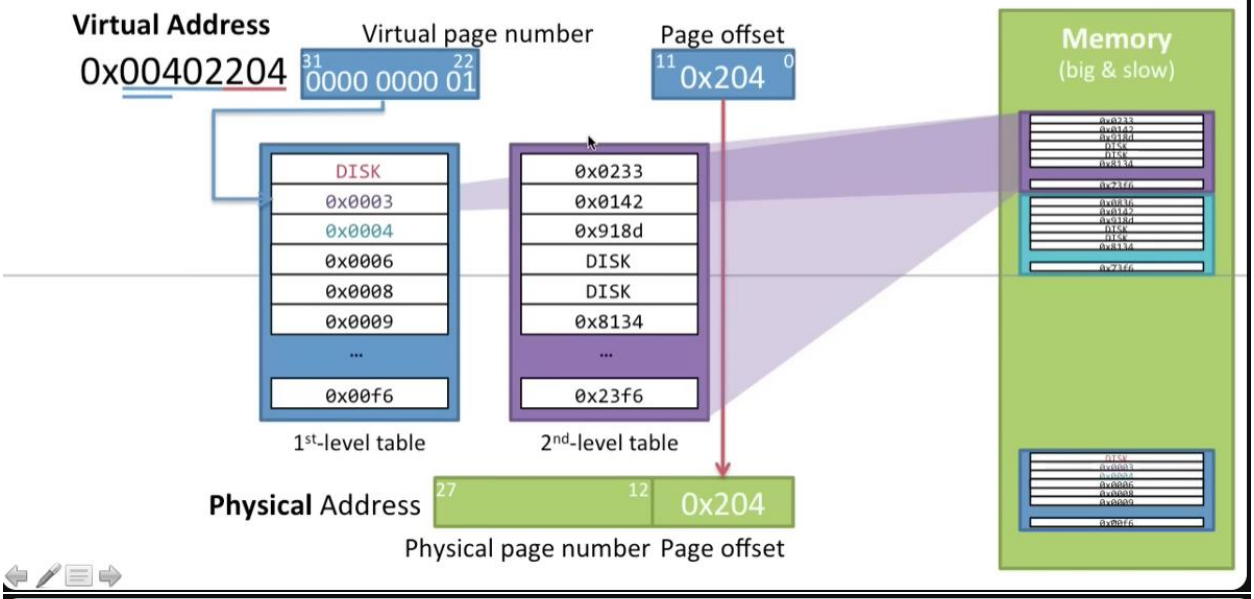


Multi-level page table translation

86

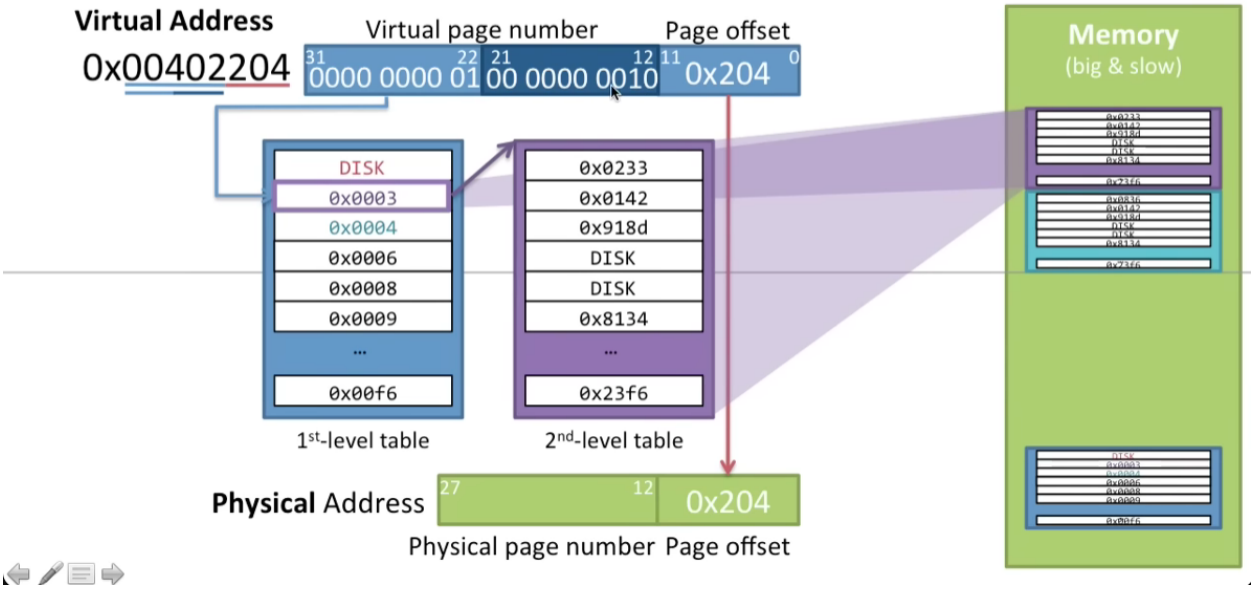


Multi-level page table translation

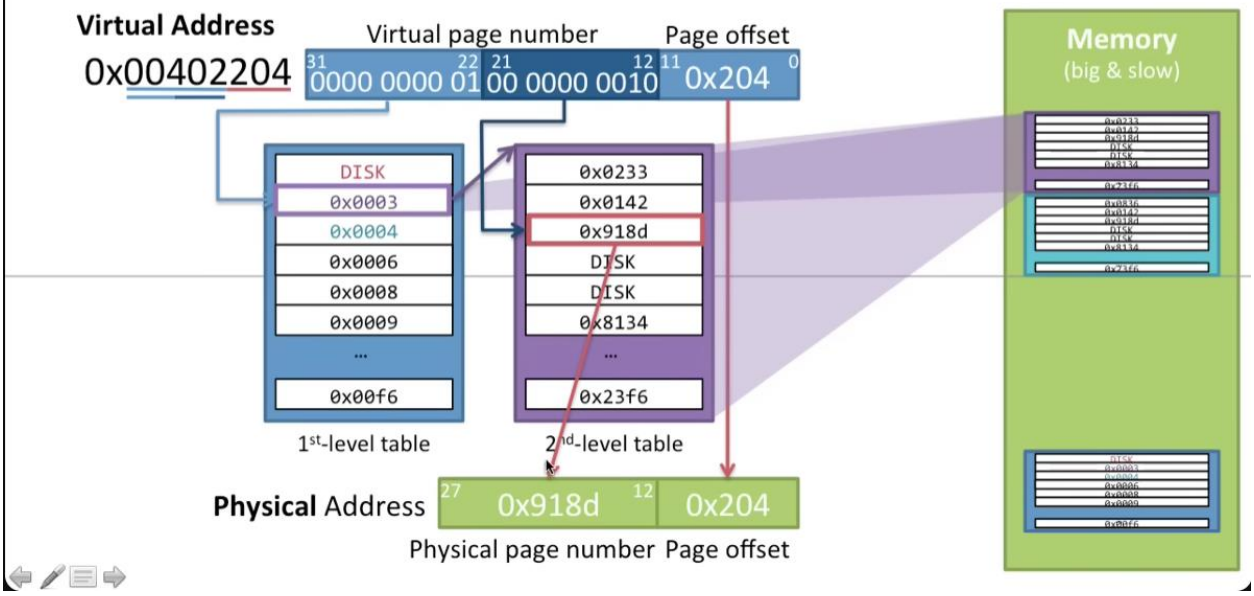


86

Multi-level page table translation

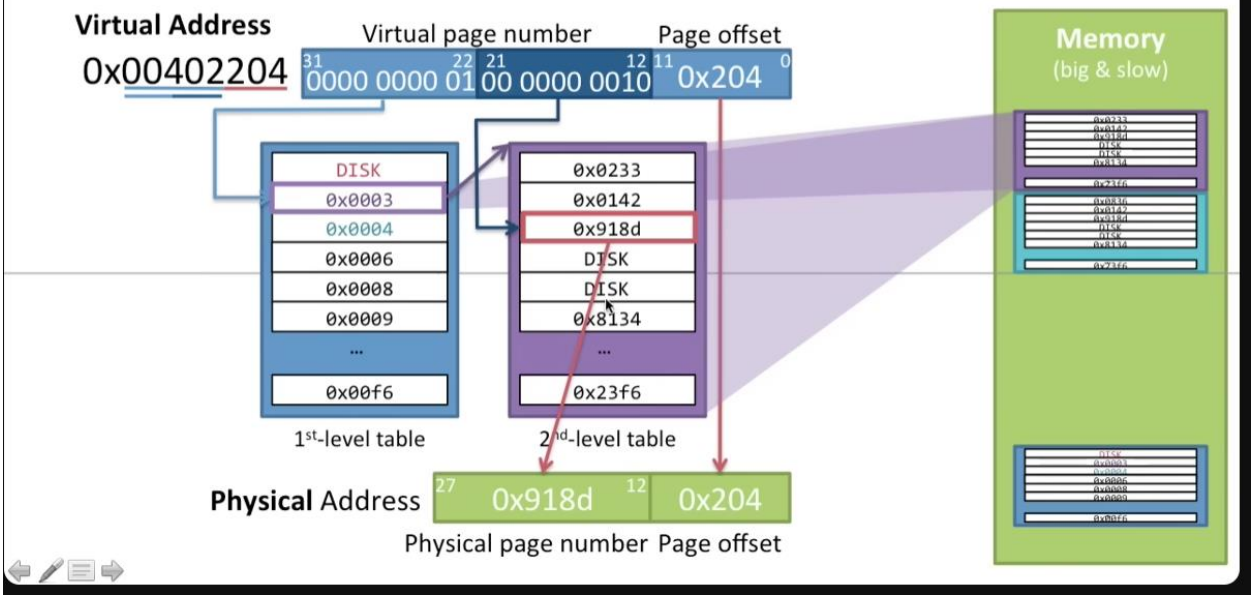


Multi-level page table translation



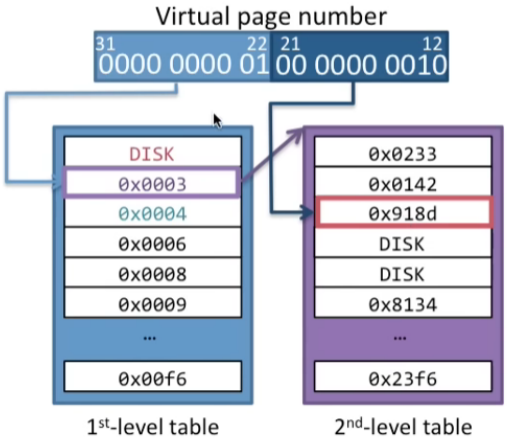
86

Multi-level page table translation



Making multi-level page tables work

- Need the **1st level page table** in memory so we can find the other pages
- Need (at least) one **2nd level page table** in memory so we can translate some program addresses
  - Need both to get an actual physical address
- How do we use them both?
  - Top 10 VA bits index into the 1st level table
  - Next 10 VA bits index into the 2nd level table



Question: Multiple applications

Q: If I am running 100 applications on my computer with a 2-level page table, how much memory do I have to keep in RAM at all times to be able to access all applications' data?

- 4kB+4kB (need the 1st-level page table and one 2nd-level page table)
- 400kB (just need the 1st-level page table for each application)
- 800kB (need the 1st- and one 2nd-level page table for each application)
- 400MB (need the full page table for each application)

A: 800kB

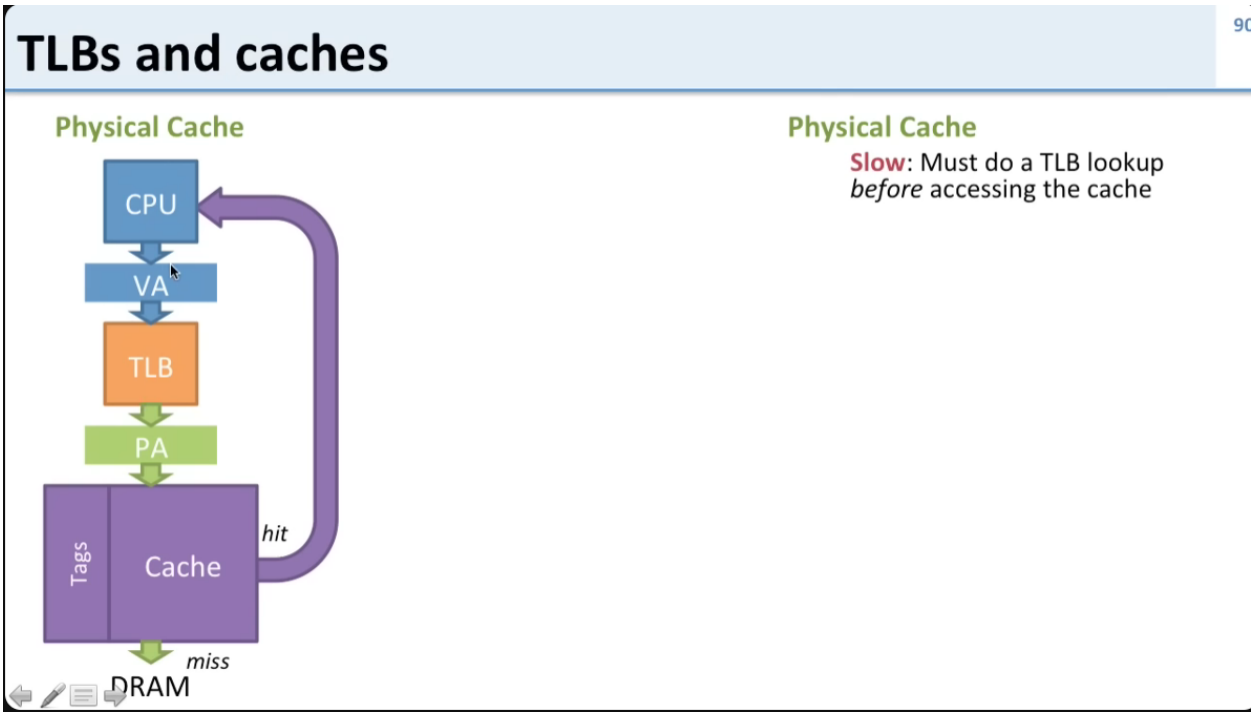
For each application we need to store its 1st-level page table (so we can find the others) and at least one 2nd-level page table so we can access some memory to do something.

If we have 100 applications, that's 100*(4kB+4kB) or 800kB.

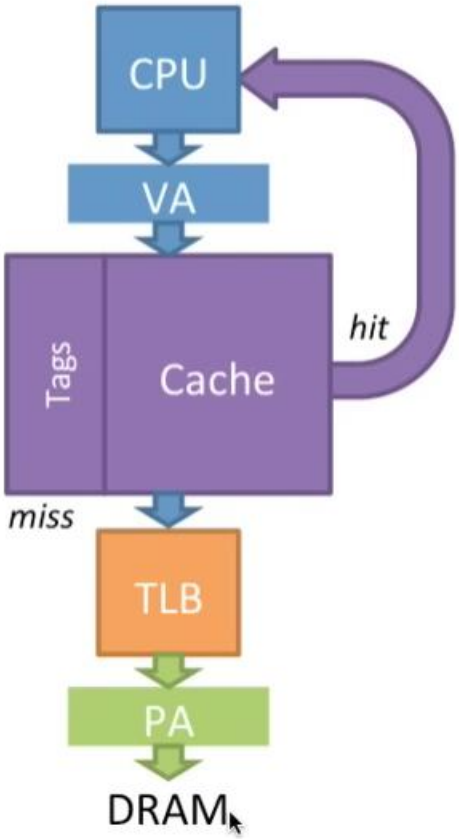
This is much better than the 400MB we would need without multi-level page tables!

Virtual Memory: 13 TLBs and Caches

https://www.youtube.com/watch?v=3sX5obQCHNA&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=13

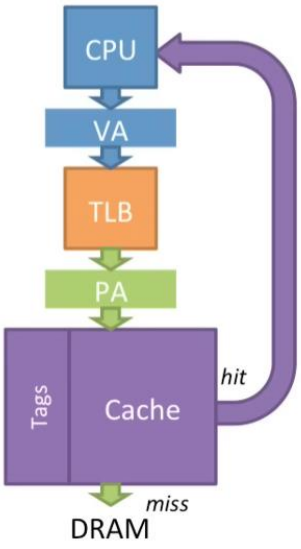


Virtual Cache

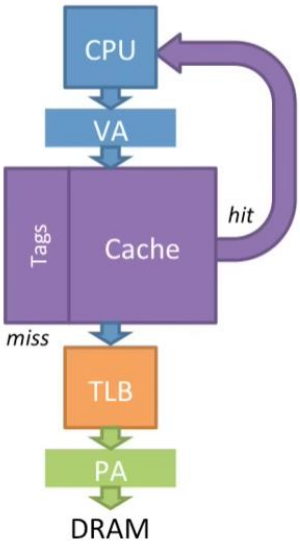


TLBs and caches

Physical Cache



Virtual Cache



Physical Cache

Slow: Must do a TLB lookup *before* accessing the cache

Virtual Cache

Fast: TLB lookups *only* when we miss in the cache

Q: Can you have two programs share a virtual cache?

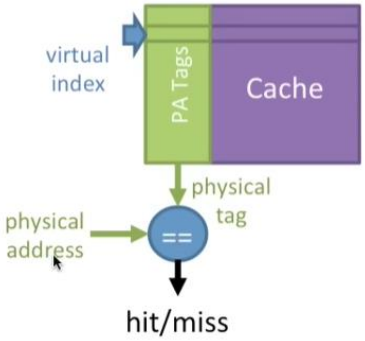
- Yes. The TLB provides protection.
- No. Each needs its own TLB.
- No. The cache is virtual so there is no way to provide protection

A: No.
A virtual cache stores data by the virtual program address. There is no translation, so VM-based protection can't keep applications apart!

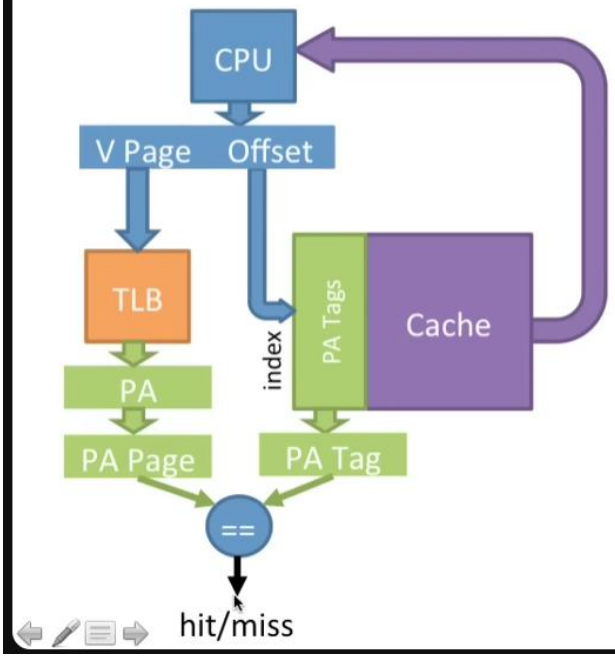
Best of both worlds: VIPT

- Can we access the cache at the same time as the TLB but still have the protection of VM?

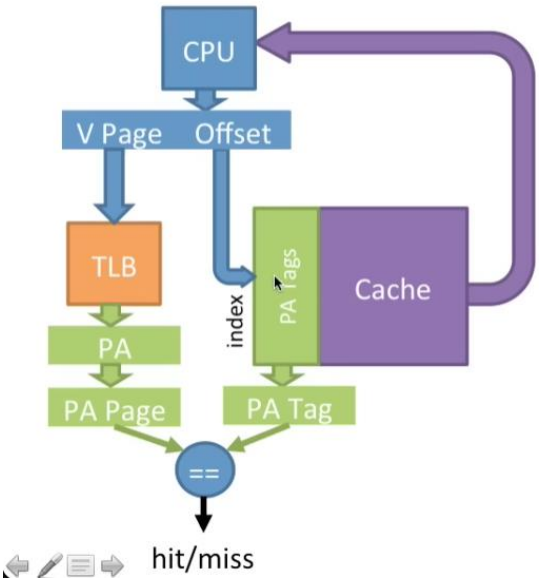
- Idea:
  - Look up in the cache with a **virtual address**
  - Verify that the data is right with a **physical tag**
- VIPT: Virtually Indexed, Physically Tagged
  - Data in the cache is indexed by the virtual address
  - But tagged by the physical address
  - We only get a hit if the tag matches the physical address, but we can start looking with the virtual address



VIPT: Virtually Indexed, Physically Tagged



VIPT: Virtually Indexed, Physically Tagged



TLB translation and **Cache lookup** at the **same time**

- Use virtual page bits to index the TLB
- Use page offset bits to index the cache
- TLB → Physical Page
- Cache → Physical Tag

Physical Page = Physical Tag → Cache hit!

Fast: look in the TLB at the same time as the cache

Safe: Cache hit only on PA match

- **But**, can only use non-translated bits to index cache (limit on how large the cache can be)
- Most processors use VIPT for L1 caches today

Virtual Memory: 14 Summary

https://www.youtube.com/watch?v=FRvzCrWcFZA&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=14

Virtual memory summary

- VM adds a level of **indirection** between the **virtual program addresses (VA)** and the **physical RAM addresses (PA)**
- This allow us to do lots of cool things:
  - Map memory to disk ("unlimited" memory)
  - Keep programs from accessing each other's memory (security)
  - Fill holes in the RAM address space (efficiency)
- But, **we have to translate every single memory access** from a VA to a PA
  - Page Tables for each program keep track of all translations
  - Use larger pages (4kB) to reduce the number of Page Table Entries (PTEs) needed
  - Fast translation via a hardware translation lookaside buffer (TLB)
- Need to combine the TLB and the cache for good performance
  - Physical caches: require translation first (slow)
  - Virtual caches: use virtual addresses (no translation: fast, but protection)
  - Virtually-Indexed, Physically-Tagged (VIPT) caches: use VA for index and PA for tag