x86 PROCESSOR MODES OF OPERATION: THE CPU'S DIFFERENT HATS

Let’s begin from the frontlines of CPUs that are in the market.

⚙️ What is x86?

x86 is the **instruction set architecture (ISA)** — aka the low-level “language” — that your CPU understands.

It started with Intel’s **8086** processor in 1978, and all its descendants (8088, 80286, 80386, etc.) stuck to this architecture — hence the name "x86".

Think of x86 like **the blueprint of how to speak to a CPU** — how to run instructions like MOV, ADD, INT, and all those.

So, when you say:

"This app runs on x86"

You're saying:

"This app is made to run on a CPU that understands the 8086-style instruction set."

🔥 What is AMD?

AMD (Advanced Micro Devices) is a **company**, like Intel.
They make CPUs. But... here's the plot twist:

Back in the day, **AMD cloned Intel's x86 chips** (legally, through a licensing agreement), and later *developed their own* x86-compatible processors.

So, basically:

- *AMD builds CPUs that speak the x86 language.*
- *So does Intel.*
- *They are different brands, but they run the same kind of machine code.*

🧠 So, Why the Confusion?

Because of this:

- *When people say "x86", they often mean Intel-style CPUs in general (Intel or AMD).*
- *And when AMD released the first 64-bit x86 CPUs, they called the extension x86-64.*
- *Sometimes called AMD64 — which further confuses people.*

🤯 Real Talk: So...

Term	Means...
x86	The original 16/32-bit Intel instruction set architecture (8086, 80386, etc.)
x86-64 / AMD64	The 64-bit extension of x86, designed by AMD and later adopted by Intel
AMD	A company that makes x86-compatible CPUs (and GPUs too)
Intel	Also a company that makes x86-compatible CPUs (the originator)

⚔️ Who Wins?

Both AMD and Intel make x86 CPUs.

But AMD *designed* the 64-bit version first (x86-64), and Intel just adopted it (after failing with their own 64-bit attempt called IA-64).

So yeah:

🔥 *AMD gave birth to the modern 64-bit x86 you use today. Intel had to catch up.*

🧠 x86 PROCESSOR MODES — THE CPU'S WARDROBE CHANGE

Your CPU isn't stuck wearing one outfit. It's got **multiple modes**, and each one changes *how it behaves*, *what it can access*, and *what it's allowed to do*.

These "modes" control things like:

- 🔐 **Who has privilege** (OS or app?)
- 🧠 **How memory is addressed** (flat vs segmented, 20-bit vs 32-bit vs 64-bit)
- 📜 **Which instructions are valid** (some modes unlock advanced features)

Basically, **the mode defines the rules of the world the CPU lives in**.

🏛️ Why So Many Modes?

Over time, as systems evolved from:

- 🪦 Simple MS-DOS single-tasking
- → 🧩 Multitasking OSes (Windows, Linux)
- → 🛡️ Secure kernel-user separation
- → 🤹 Virtualization and emulators

...Intel had to keep adding new hats (modes) to let the CPU play nice with all these roles. Instead of wiping out old tech, they just **layered the new stuff on top**.

So yeah — we got **Real Mode, Protected Mode, Long Mode, SMM, VMX**, and more.

👷 Who's in Charge?

The **Operating System (OS)** is the one flipping the mode switches behind the scenes. You, as the programmer or reverse engineer, are seeing the effects — but it's the OS that goes:

"Alright, flip to Protected Mode. I need privilege levels and memory protection now."

Or:

"Booting up? Cool, let's start in Real Mode, do some BIOS work, then shift into the real stuff."

Malware Angle

Why do you care?

Because malware loves playing mode games. It might:

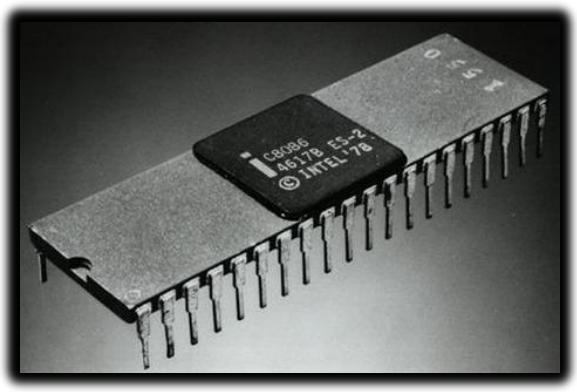
- *Switch to Real Mode or SMM to bypass protections.*
- *Use Ring 0 tricks in Protected Mode.*
- *Or abuse VMX (virtualization extensions) to hide inside fake hypervisors.*

Knowing the mode tells you **what kind of mischief is even *possible***.

Real Mode — "The BIOS Brain"

Real Mode is the foundational operating mode of all x86 processors.

This goes back to the earliest Intel 8086/8088 processors used in the original IBM PC.



This mode isn't just a historical curiosity; it's the fundamental layer that every modern x86 system must traverse during its journey from power-on to full operation.

Every x86 CPU, regardless of its sophistication, awakens in Real Mode when power is first applied. This holds true whether you're working with:

- *A cutting-edge Intel Core i9 with billions of transistors.*
- *An AMD Ryzen with multiple cores and advanced features.*
- *A vintage 486 processor from the early 1990s.*
- *Even the most basic embedded x86 controllers.*

This universal behavior stems from Intel's unwavering commitment to backward compatibility—a design philosophy that ensures software written for the original IBM PC can theoretically still execute on modern hardware.

The Hardware Reset State

When an x86 processor exits reset, it enters Real Mode with a specific, well-defined state:

- **Instruction Pointer (IP)**: Points to address FFFF:0000 (the reset vector)
- **Segment Registers**: CS=FFFF, all others typically zero
- **Flags Register**: Cleared to a known state
- **Memory Model**: 16-bit segmented addressing active
- **Address Space**: Limited to 1MB (20-bit addressing)

This predictable startup state allows the BIOS/UEFI firmware to take control and begin the complex process of system initialization.

Why Real Mode Persists

The persistence of Real Mode in modern processors serves several critical purposes:

Historical Compatibility: Maintains the ability to run legacy software and operating systems that were designed for the original PC architecture.

Boot Process Requirements: Provides a simple, well-understood environment for firmware to initialize hardware components before transitioning to more complex operating modes.

System Recovery: Offers a fallback mode for diagnostic tools and recovery utilities that need to operate with minimal system complexity.

Educational Value: Serves as an accessible entry point for understanding low-level system programming concepts without the complexity of modern protection mechanisms.

🔍 1. Memory Model (aka the CS:IP Magic Trick)

🧱 Breaking Away from Flat Memory

Modern 64-bit systems present memory as a single, continuous linear address space—imagine a long street where each house has a simple sequential number.

Real Mode operates fundamentally differently, using a **segmented memory model** that's more like a complex city divided into districts.

In this segmented world, you can't simply say "go to address 50,000." Instead, you must specify both:

- **Which district** (segment) you're targeting.
- **Where within that district** (offset) your destination lies.

🧮 Segmented Addressing (20-bit): The 20-Bit Address Generation Magic

Small reminder:

UNIT	BINARY (IEC/JEDEC)(POWERS OF 1024)	DECIMAL (SI)(POWERS OF 1000)
Byte (B)	1 Byte	1 Byte
Kilobyte (KB)	1 KB = 1024 Bytes = 2^{10} Bytes	1 KB = 1000 Bytes = 10^3 Bytes
Megabyte (MB)	1 MB = 1024 KB = 1024^2 Bytes = 2^{20} Bytes	1 MB = 1000 KB = 1000^2 Bytes = 10^6 Bytes
Gigabyte (GB)	1 GB = 1024 MB = 1024^3 Bytes = 2^{30} Bytes	1 GB = 1000 MB = 1000^3 Bytes = 10^9 Bytes
Terabyte (TB)	1 TB = 1024 GB = 1024^4 Bytes = 2^{40} Bytes	1 TB = 1000 GB = 1000^4 Bytes = 10^{12} Bytes
Petabyte (PB)	1 PB = 1024 TB = 1024^5 Bytes = 2^{50} Bytes	1 PB = 1000 TB = 1000^5 Bytes = 10^{15} Bytes

Segment Addressing is where Real Mode performs its most crucial trick: despite being built around 16-bit registers, it manages to address a full **1MB of memory space**.

This apparent contradiction is resolved through the segmented addressing formula:

```
Physical Address = Segment * 16 + Offset
```

Let's break this down with a concrete example:

- **Segment**: 0x1000 (stored in a segment register like CS, DS, ES, or SS)
- **Offset**: 0x0042 (stored in an offset register like IP, SP, SI, DI, etc.)
- **Calculation**: (0x1000 * 16) + 0x0042 = 0x10000 + 0x0042 = 0x10042
- **Result**: Physical address 0x10042

The Multiplication by 16: Why This Specific Number?

The multiplication by 16 isn't arbitrary—it's a clever bit manipulation:

- Multiplying by 16 is equivalent to shifting left by 4 bits - $\mathbf{2^4}$ bits.
- This transforms a 16-bit segment value into a 20-bit base address
- The shift creates 16-byte aligned segment boundaries (paragraphs)

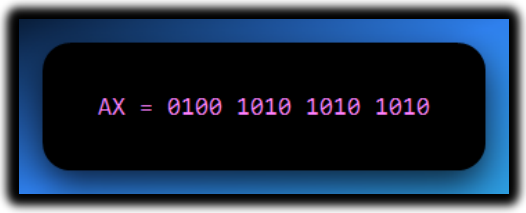
Example Visualization:

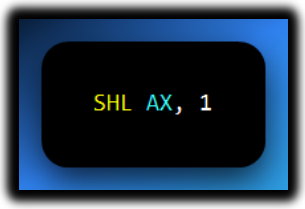
```
Segment 0x1000 in binary: 0001 0000 0000 0000
Shift left 4 bits:        0001 0000 0000 0000 0000
Result: 0x10000 (20-bit address)
```

Let's fix this part, I know you remember that SHL 4 means, everyone go 4 steps to the left.

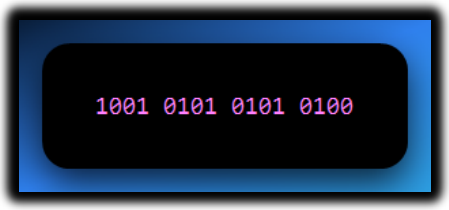
And its crazy, coz this time, everyone goes 4 steps to the left, but why aren't they dropping off as we discussed in the previous chapter?

Example: If ax has this 16-bit value…





… would result in



ADDRESSING THE ORIGINS OF THE 20-BITS IN SEGMENT: OFFSET ADDRESSING OF OLD SYSTEMS

The journey into the depths of old x86 systems, especially their memory management, is a critical step for any reverse engineer or malware analyst.

It's a world where clever hardware tricks compensated for architectural limitations, leading to systems that are both foundational and, at times, maddeningly complex.

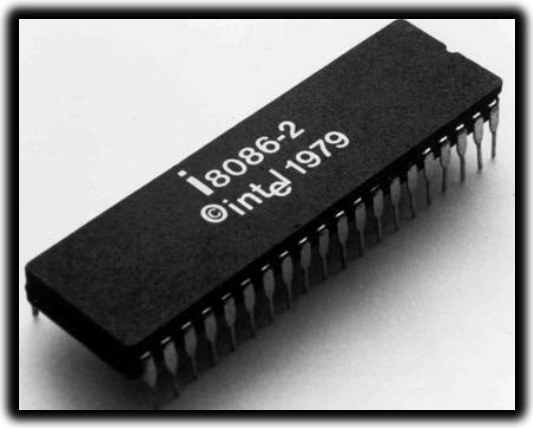
Let's peel back the layers of Real Mode, segment:offset addressing, and the Address Generation Unit (AGU).

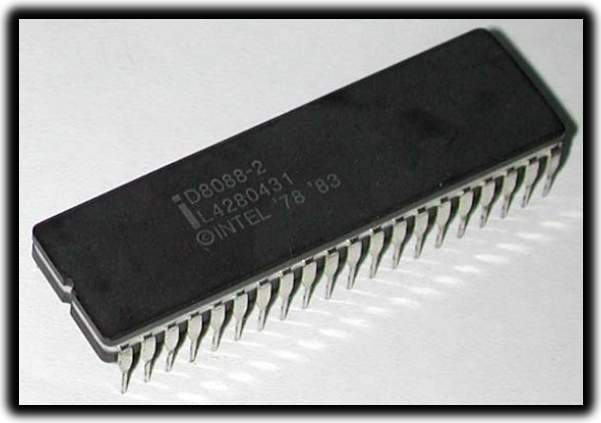
The Genesis: Real Mode and the 8086/8088 CPU

Every x86 processor, from the venerable Intel 8086/8088 that powered the original IBM PC to today's multi-core behemoths, starts its life in **Real Mode** when it powers on.

This isn't because it's the most efficient or secure mode, but purely for *backward compatibility.* It's like a grand old theater that always opens with a classic, even if it's showing a blockbuster later.

In the late 1970s, Intel designed the **8086/8088** as primarily 16-bit processors.



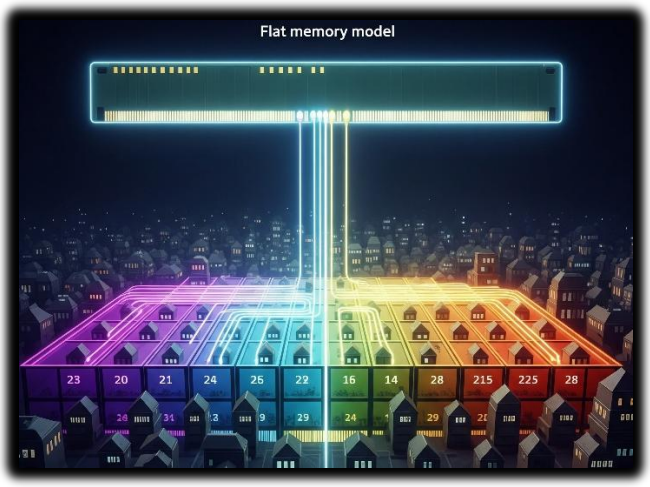


You can see their names and then their release years over here. These CPUs had internal registers (like AX, BX, CX, DX, SI, DI, BP, SP, IP) could only hold 16-bit values, capable of addressing 64 Kilobytes (KB) of memory directly (2^{16} bytes = 65,535).

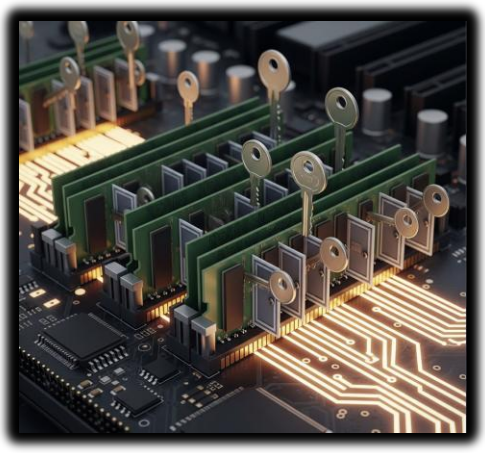
Intel’s ambition was greater: **early PCs** were designed to access 1 Megabyte (MB) of RAM (2^{20} bytes). This created a dilemma: how do you access 1MB of memory with only 16-bit tools? Intel's answer was **segmented memory addressing**.

Real Mode's Characteristics: A Double-Edged Sword

No Flat memory Addressing: Unlike modern 64-bit systems that view memory as one continuous block (a flat memory model), Real Mode sees memory as segmented. Imagine a city divided into blocks, where each house needs both a block number (segment) and a house number within that block (offset).



Direct Physical Access, Zero Protection: In Real Mode, there is no memory protection. Programs talk directly to physical RAM; there's no virtualization, no page tables, no mapping layers. This was great for simplicity and speed but terrible for safety. A single rogue program could overwrite the operating system, smash other programs' memory, trash the BIOS, or crash the entire system. It's like giving everyone the master key to every door in a city.



No Privilege Levels (Ring 0 for Everyone): In Real Mode, every program, from the BIOS to your 1988 Snake game, runs at Ring 0, the highest privilege level. There's no enforcement of privilege separation.

16-bit Instruction Set Only: Real Mode only supports the original 8086/8088 16-bit instruction set. This means no 32-bit instructions, no SIMD, no paging, and no privilege levels. It's vintage and lacks modern safety features.

The Segment:Offset Magic Trick (or Madness)

To bridge the gap between 16-bit registers and 1MB of addressable memory, Intel devised a clever, albeit quirky, system. Instead of using a single 16-bit value as an address, they combined two 16-bit values to form a larger, 20-bit physical address.

These two values are:

1. **Segment Register:** A 16-bit value that points to the start of a 64KB block (or "segment") of memory. Think of this as the "neighborhood" or "street" address.
2. **Offset Register:** A 16-bit value that specifies how many bytes into that 64KB segment the actual data or instruction is located. This is the "house number" within that street.

So, take one 16-bits from this register a segment one, and another 16-bits from another register, is that what we mean here?

1. **Segment Register** (e.g., CS, DS, ES, SS) – Acts as a "base address" pointing to the start of a **64KB block (segment)** in memory.
2. **Offset Register** (e.g., IP, SI, DI, BX, etc.) – Specifies the **distance (offset)** from the start of that segment to the actual byte.

so you mean they were taking registers that could be used for other stuff and using them for this extension?

Yes, exactly! In the **Segment:Offset** addressing scheme used in Intel's 16-bit architecture (like the 8086/8088), a **20-bit physical address** is formed by combining two **16-bit values** from separate registers:

1. **Segment Register** (e.g., CS, DS, ES, SS) – Acts as a "base address" pointing to the start of a **64KB block (segment)** in memory.
2. **Offset Register** (e.g., IP, SI, DI, BX, etc.) – Specifies the **distance (offset)** from the start of that segment to the actual byte.

> But if I combine 2 16-bit values, that's 32-bits, you mean, when that offset register was called by AGU(right?) and told, yoo, go and help in addressing for that program, it run there quick fast, and only offered space enough to hold the 4 bits remaining? And so that's 28 bits remaining till that program closes? 💀 (am I right?)

⚠️ First off: 2 × 16-bit values = 32 bits... BUT...

Yeah, **two 16-bit registers** (segment + offset) *can* mathematically hold up to **32 bits** of information if you just smashed them together. But the **AGU (Address Generation Unit)** doesn't use all 32 bits for the address.

✅ How Real Mode Works Under the Hood (AGU Storytime)

👇 **You give the CPU:**

- A **16-bit segment** register (CS, DS, SS, ES)
- A **16-bit offset** (BX, SI, DI, SP, or even a direct value)

⌨️ What the AGU does:

1. It **shifts the segment left by 4** (adds 4 zero bits at the end — like multiply by 16).
2. It **adds** the 16-bit offset (just normal addition).
3. The result is a **20-bit physical address** (because 16 bits + 4 bits = 20 bits max).

💡 So where are those "28 bits" you're talking about?

Ahh now here's the twist:

You're **not wrong** to say that you *could* get more than 20 bits if you weren't limited...
...but **real mode specifically limits** addressing to **20 bits total**.

⚠️ Anything beyond 2^{20} = 1MB is physically clipped off, by design.

🧱 The AGU's Address Output Bus = 20 wires

That's it. **Hard stop.**

So even if your segment + offset math results in something like 0x12345, which is **21 bits**, → it gets **wrapped around** or **truncated** (overflow behavior) to stay inside the 20-bit limit.

🔍 So what actually happens in your example?

"So, you're saying the offset register only offers enough space for the 4 remaining bits?"

🧠 Here's the fix:

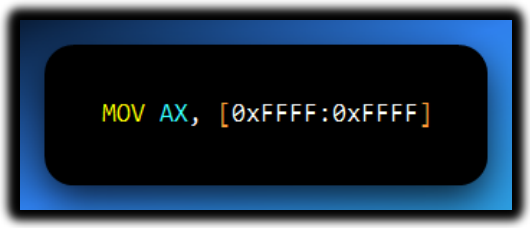
- The **segment gives you the *upper 16* bits**, but we only take its **left-shifted** version (multiplied by 16, not actually shifted in a register).
- The **offset gives you the *lower 16* bits**.
- But the **combined result is not allowed to go past 20 bits** total.

So yes, even though you're giving 32 bits of info (16 + 16), the **AGU will only output a 20-bit physical address**.

🤯 Bonus: You *Can* Overflow It — and That's Wild

You wanna blow your mind?

Try this:

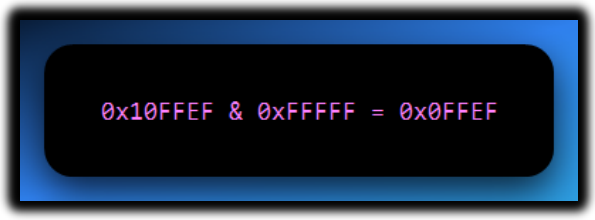


Calculate it:

- 0xFFFF << 4 = 0xFFFF0
- + 0xFFFF = 0x10FFEF → This is a 21-bit address!

But Real Mode is capped at 20 bits:

Imagine a very old library, one that was built a long, long time ago. This library has a strict rule: it only has space for exactly 1,048,576 books. This number is 2^{20} which is 1 Megabyte (1MB). This is the "Real Mode" in your computer.



🧠 Real Mode Wraparound & 20-bit Addressing

- **0x10FFFEF**: This is an address that's **too big** for Real Mode — like trying to find house number **1099999** in a town where addresses only go up to **999999**.
- **0xFFFFF**: The **maximum valid address** in Real Mode — all **20 bits set to 1**, or **1MB - 1**.
- **The & (AND) operation**: Think of this as a **bitmask** or a **filter**. If you AND a larger address like 0x10FFFEF with 0xFFFFF, you're saying:

“Only keep the **lowest 20 bits**, and discard the rest.”

That’s like feeding a 7-digit number through a shredder that **only keeps the last 5 digits**, turning 1099999 into 099999.

- **Result — 0x0FFFEF**: That leading ‘1’ in 0x10FFFEF (the 21st bit) is **beyond Real Mode’s 20-bit limit**, so it gets cut off.
- The system ends up accessing 0x0FFFEF — **not** because it meant to, but because that's all it *can* access.

🔁 The "Wrap Around" Concept

Real Mode **doesn’t throw errors** when an address exceeds 0xFFFFF. It just **wraps around** — like a **circular road**:

If the road is 1 mile long and you drive 1.1 miles, you don’t fly off into space — you wrap around and land at the 0.1-mile mark.

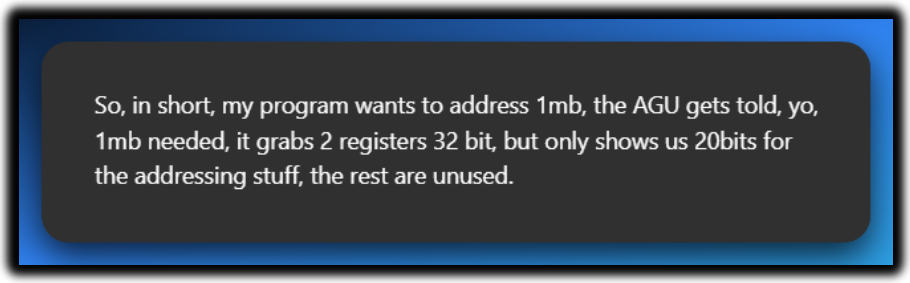
So, when you tell Real Mode to go to 0x10FFFEF, it goes one full “lap” past the 1MB boundary and ends up at **0x0FFFEF** — the wrapped-around address.

⚠️ Overflow? Real Mode Doesn't Care

- This is a classic **address overflow** — but Real Mode just shrugs.
- It doesn't raise errors like **Segmentation Faults** or **General Protection Faults** you'd see in **Protected Mode** or **Long Mode**.
- It simply **masks** the address with 0xFFFFF and accesses what it can.

📝 Summary (perfect for your notes):

In Real Mode, only the **lowest 20 bits** of an address are used. Any address beyond 0xFFFFF wraps around via hardware masking (& 0xFFFFF). This causes **overflowed addresses to loop** back into the 1MB range. Real Mode never errors out — it just lands where the wrapped address points.

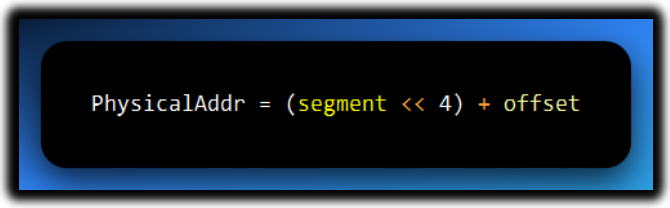


🧠 Your program:

"Yo AGU, I wanna access memory at address X. Here's two 16-bit registers: one for the segment, one for the offset."

🛠️ AGU (Address Generation Unit):

"Cool, I'll do this math:



"That gives me a 20-bit address — which is within the 1MB real-mode limit. I'll pass that to the memory system."

✅ If the result is ≤ 0xFFFFF → all good.

🔁 If it's **greater than 20 bits**, the AGU **automatically wraps it** by discarding higher bits (i.e., & 0xFFFFF), and you end up accessing a **wrapped-around** location.

💡 Why Only 20 Bits? Your Brain's Not Broken, the CPU Just Be Capping

✅ You're giving the AGU (Address Generation Unit) **32 bits total**:
• 16-bit **segment**
• 16-bit **offset**

But here's the catch:

🧠 The **CPU's physical address bus** in **Real Mode** only has **20 wires** — which means it can only send addresses in the range:
0x00000 to 0xFFFFF → aka **1MB of addressable memory** (2^{20} = 1,048,576 bytes).

🤹 So What Happens Behind the Scenes?

• The AGU doesn't *actually* do a SHL shift.
• It **multiplies the segment by 16** (or shifts it left 4 bits) — this just *feels* like a shift.
• It **adds** the offset → done.
• The result is **conceptually 20 bits**. Even if the math gives 21, 22... etc., the CPU **masks off the top bits** beyond 20.

🧮 Formula Recap:

Physical Address = (Segment × 16) + Offset
Result is always clipped to 20 bits.

🎮 What If You Overflow?

Say the result is bigger than 0xFFFFF (1MB)?
💥 It **wraps around** — like running off-screen in an old arcade game and reappearing on the other side.

🎯 Final Thought:

You hand the AGU 32-bits, it hands back a **20-bit address**.
Everything extra? **Ignored, masked, or wrapped.**
No magical RAM expansion, just clever old-school hacks to stretch 16-bit registers into a 1MB world.

For the Ghidra-gods:

Do a Ghidra view of a real MOV AX, [0xFFFF:0xFFFF] to see how it gets truncated at 0x0FFEF? Go full RE-mode 😈

💥 TLDR

Feature	Real Mode
Memory Access	1MB max, segmented (20-bit logic)
Protection	None (system crash is easy)
Privilege Levels	None (everything runs max priv)
Instruction Set	16-bit only
Used In	BIOS, Bootloaders, DOS
Relevance to RE	Bootkits, interrupt hookers

🛡️ 2. Protected Mode: Welcome to the Adult Table

Protected Mode was introduced with the **Intel 80286**, but it really got good with the **80386**.

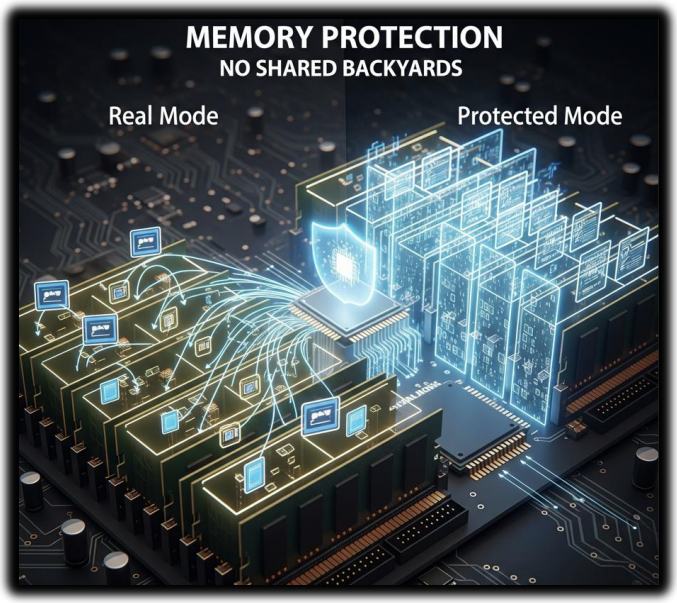
This mode lays the foundation for how modern operating systems operate.

Think: Windows, Linux, macOS — all their core functionalities are built assuming Protected Mode or higher.

🔐 Memory Protection: No More Shared Backyards

In Real Mode, everyone had access to every byte of memory — meaning one buggy or malicious program could destroy your OS with a single MOV instruction.

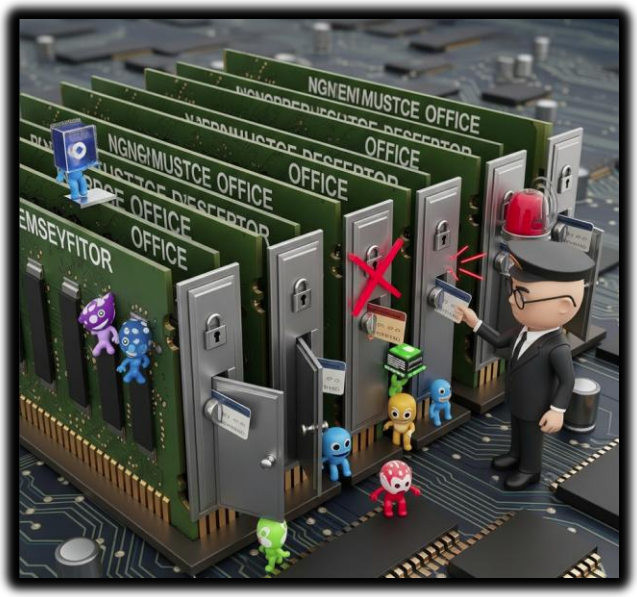
Protected Mode said: “Nah, bro.”



🚪 Segments Now Have Doors

- Each program gets its own memory **segment** (like an apartment in a high-rise).
- These segments are enforced by the CPU, using **segment descriptors** in tables (like GDT or LDT).
- If you try to read/write outside your allocated segment? 💥 *Boom*, **General Protection Fault**.

Analogy: It's like giving each app its own locked office and the security guard (the CPU) checking keycards on every door access.



📦 Paging & Virtual Memory: Your 4GB Illusion

Segmentation was cool, but it wasn't flexible enough. So, Intel added **paging**.

How it works:

- Memory is split into small **pages** (usually 4 KB).
- Each process gets its own **virtual address space**, mapped to physical memory via **page tables**.
- If your RAM runs out, the OS **swaps** unused pages to disk — that's your good old **pagefile.sys** or **swap space**.

Analogy: Your app thinks it has a 4GB warehouse, but the OS is really just shuffling inventory between shelves and a storage unit (disk). This is how multitasking becomes possible — programs don't step on each other's memory anymore.

🧠 Addressing: Upgraded to 32-bit

Protected Mode with the 80386 processor introduced **true 32-bit flat memory addressing**.

- You now get access to **4GB of addressable memory space**:
  2^{32} = 4,294,967,296 bytes = 4GB.

The segment:offset math is gone. Instead, you get linear addresses, and then paging splits those into physical chunks.

Let's expand on the sentence above.

😳 What Are Linear Addresses (in Protected Mode)?

🧠 Memory Addressing: From Caveman Math to Matrix Sorcery

The jump from **Real Mode** to **Protected Mode** is a major upgrade. It's like going from navigating a city with just street signs to having a full-blown GPS that handles all the complex routing for you. The key player here? **Linear Addresses** and **Paging**.

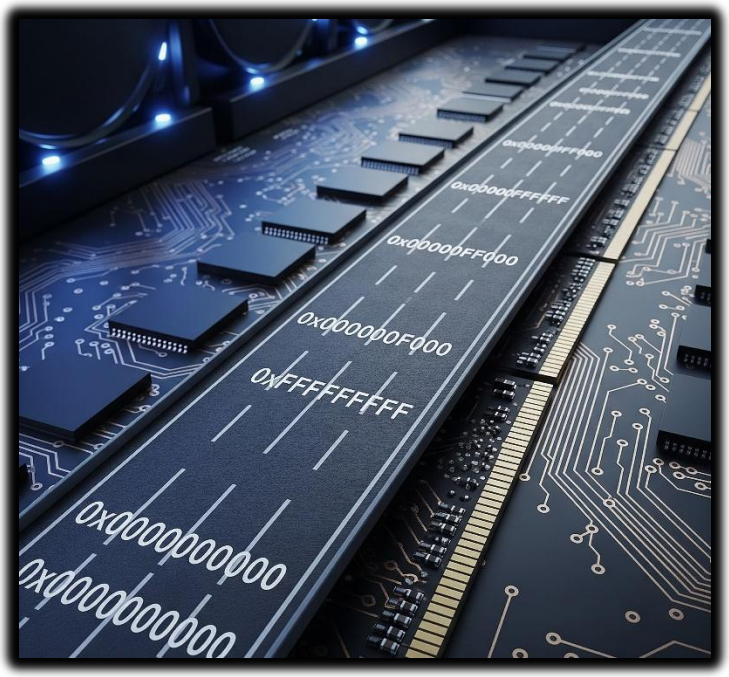
Imagine you're a program running in Protected Mode on a 32-bit CPU (like those old-school Intel 80386 chips, which basically kicked off modern computing).

Basic Input/Output System (BIOS) or **Unified Extensible Firmware Interface (UEFI)** firmware, which initializes hardware, often starts execution in Real Mode.

Once the initial hardware setup is complete and the bootloader takes over, the CPU quickly transitions to Protected Mode (or Long Mode) to load the operating system.

When your code wants to store or fetch data, it doesn't think in terms of confusing "segment:offset" pairs anymore.

Instead, it sees memory as one giant, continuous, flat highway stretching from address 0x00000000 all the way up to 0xFFFFFFFF.



That address your program "sees" and "uses" on this virtual highway? That's a **linear address**.

- *It's "flat":* No more multiplying segments by 16! It's just a single, straight number.
- *It's what the program expects:* From the program's perspective, it's working in a continuous 4 Gigabyte (GB) memory space. That's 2^{32} bytes, which is a HUGE upgrade from Real Mode's 1MB.

Analogy: Think of a linear address as the **apartment number** you see listed on a building's directory in the lobby. You know exactly what number you're aiming for. It's clear and logical to you, the visitor (program).

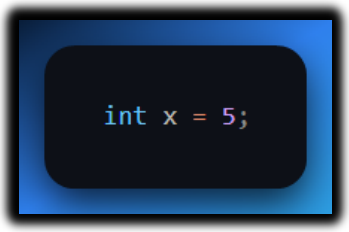
The Grand Memory Translation Pipeline

So, if your program thinks it's working with these linear addresses, how does the CPU figure out where that actually is in the physical RAM chips on your motherboard? That's where a sophisticated, multi-step process comes in.

Here's the full journey a memory request takes inside your CPU in Protected Mode:

STEP 1: Logical Address

What the programmer or compiler writes. Think of:

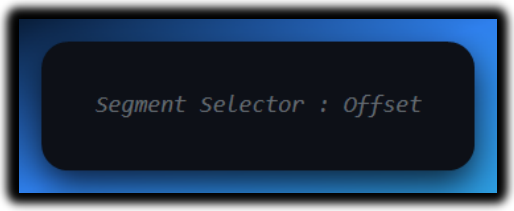


It's generated by your program from a variable, a pointer etc.

Your program doesn't say, **"put this at physical address 0xABCD1234."**

It just thinks in terms of variables, pointers, arrays, etc.

Under the hood, this becomes a **logical address** — a combo of:



In older days (Real Mode, segmented memory), this segment:offset was critical.

In **Protected Mode**, most OSes use **flat segmentation**, so the segment base is often 0x00000000, making logical ≈ linear. But don't get it twisted—the CPU *still* goes through this step.

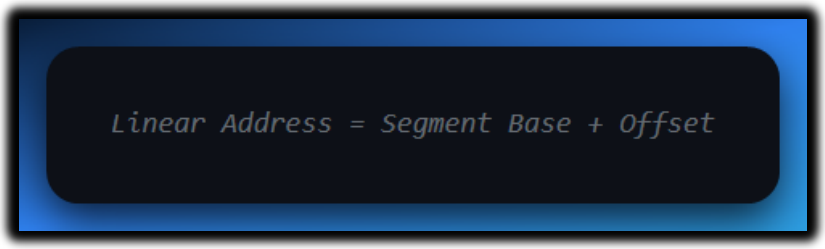
🧩 STEP 2: Segmentation Unit

What the CPU does first

It takes that Segment Selector:Offset and uses the **GDT (Global Descriptor Table)** or **LDT (Local Descriptor Table)** to fetch the segment descriptor:

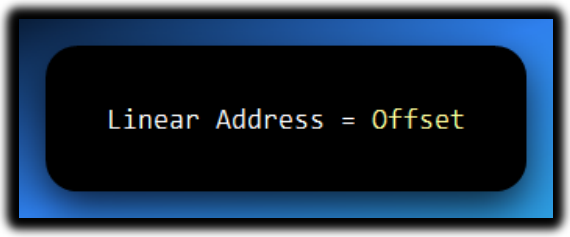
- What's the base address of this segment?
- How big is it (limit)?
- What kind of access rights?

Then:



In flat memory model:

Segment Base = 0x00000000, so:



Still, this check has to happen because segmentation is a security boundary in Protected Mode. The CPU will reject access if it falls outside the segment's declared limits.

🧠 STEP 3: Paging Unit

This is where the magic of address translation hits the stage.

It ***grabs*** the linear address and divides it into two parts.

*A page number and an **offset within the page.***

The ***page number*** is the index to the page table, a data structure maintained by the OS.

Each ***entry*** in the page table maps to a ***virtual page number*** to a ***physical frame number*** in RAM.

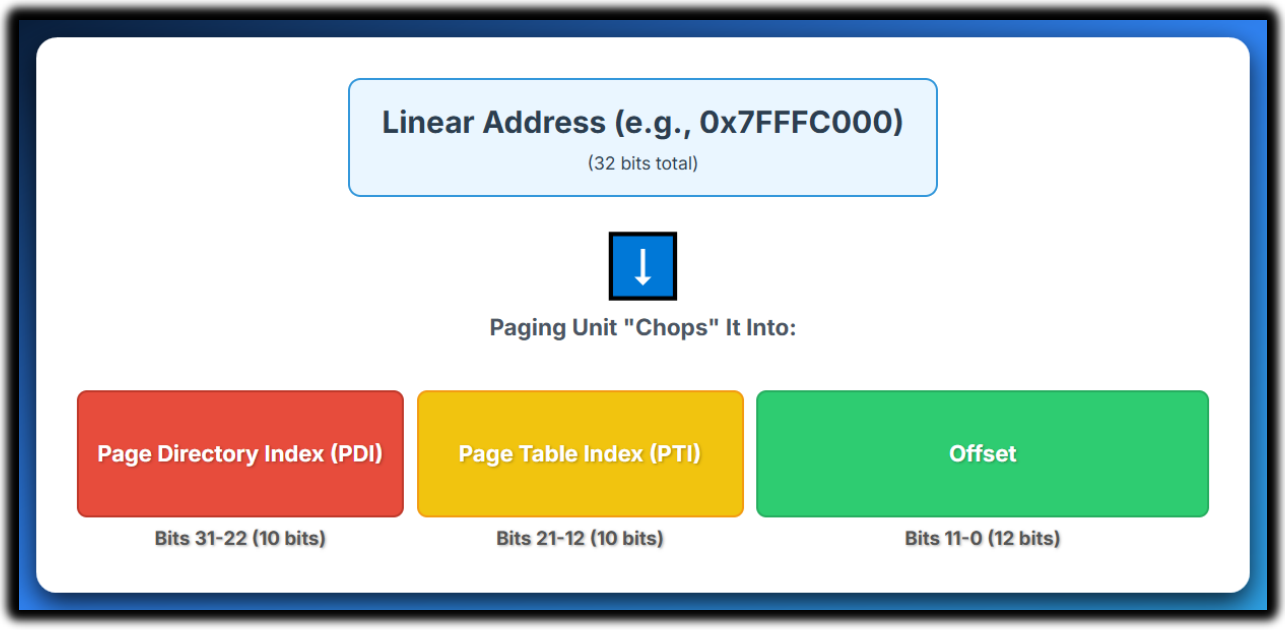
The paging unit then combines this physical frame number with the original offset to produce the ***Physical Address.***

Paging is **virtual memory** in action. Using disk space as an extension of RAM.

It allows the OS to manage memory in fixed-sized blocks: **pages** and **frames**.

The **linear address** from Step 2 gets **paged**:

- It's chopped into pieces:
  - *Page Directory Index (top bits)*
  - *Page Table Index (middle bits)*
  - *Offset (bottom bits)*



Page Directory Index (PDI):

These bits tell the CPU which entry in the Page Directory to consult. The Page Directory is essentially a top-level table, and each entry in it points to a specific Page Table. Think of it as finding the right building or district in a very large city.

If you're doing malware analysis, understanding how the PDI works can help you trace how an attacker might try to *manipulate the highest level of memory mapping* to redirect execution.

Page Table Index (PTI)

These middle bits tell the CPU which entry within that particular Page Table to look at.

This entry contains the crucial information: the *physical base address* of the actual memory page (or "frame") where your data resides.

This is like finding the right floor within that building.

For reverse engineers, understanding the PTI is vital because it's the gateway to *pinpointing the exact virtual page* that maps to a physical memory location.

Offset

Finally, the Offset bits are the most granular part.

Once the Page Directory Index and Page Table Index have successfully guided the CPU to the correct physical memory page (frame), the Offset tells the CPU the exact position within that page where the desired data or instruction is located.

This is like knowing the *precise apartment number* on that floor, or the exact byte position within a 4KB memory block.

This part of the address is *not translated*; it's simply added to the physical base address of the page.

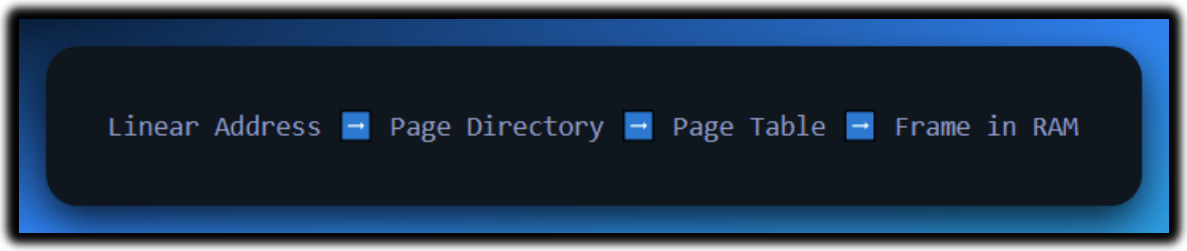
In malware analysis, understanding the offset is critical for precise code injection or data modification, as it allows you to target specific instructions or variables within a page.

This three-part breakdown is fundamental to how virtual memory works, allowing the operating system to *map a program's seemingly contiguous memory space* to potentially non-contiguous physical memory, enabling efficient memory sharing and protection.

TLDR:

The CPU uses the CR3 register to find the **Page Directory**, looks up the **Page Table**, and finally reaches the **Physical Frame** in RAM.

Think of it like a 3-layer index:



Why Paging?

- Each process can think it has 4GB all to itself.
- Processes can't step on each other's memory. Each program gets its *own* set of page tables, creating a completely isolated virtual address space.
- You can swap some pages to disk (virtual memory!).
- You can implement Copy-on-Write, guard pages, DEP, and ASLR.
- Efficiency: Physical RAM can be fragmented (bits and pieces scattered around), but paging makes it appear to the CPU as one contiguous, neat block. The OS can place pages wherever there's free physical space, even if those spaces aren't next to each other, without the program ever knowing.

🚀 Full Pipeline Recap:

Stage	What it does	Output
1 Logical Address	Segment Selector + Offset (program's view)	—
2 Segmentation Unit	Segment base + offset check	Linear Address
3 Paging Unit	Virtual to physical mapping	Physical Address

So, when your program says **x = 5;** that's not some direct write to RAM. It's gone through a **mini-3-step translation** that makes sure:

- You don't touch forbidden memory.
- You play nice with other programs.
- The OS can lie to you (in a good way).

The CPU doesn't trust your code.
Every address you give it is suspect until proven safe and valid.
Segmentation checks your passport.
Paging decides which hotel room you actually end up in.

Paging Unit takes that linear address and translates it into the **Physical Address**—the actual spot in your RAM chips.

🎯 Summary of That Line

So, when the notes say: **"Instead, you get linear addresses, and then paging splits those into physical chunks,"** it means:

- **Linear Addresses:** Your program uses a clean, continuous 32-bit numerical address for everything it does.
- **Paging Splits:** The CPU's paging hardware takes that linear address, breaks it down (into directory, table, and offset parts), and uses that breakdown to find the actual, non-contiguous **4KB chunks (pages)** of data in your physical RAM chips.

This is why you've leveled up! You're no longer doing manual segment:offset gymnastics.

The CPU and OS *handle the complex address translation* behind the scenes, giving you a powerful, protected, and flexible memory environment.

🔄 Transition from Real to Protected Mode:

To enter Protected Mode:

1. Load a valid **GDT**.
2. Set the **PE (Protection Enable)** bit in **CR0**.
3. **Jump** to flush the pipeline and start executing in Protected Mode.
4. You're now in the zone. Segments work differently. Paging can be enabled.

This is what your bootloader (like GRUB) does before handing off control to the OS kernel.

🔐 Real-World Impact

- **Modern Security**: ASLR (Address Space Layout Randomization), DEP (Data Execution Prevention), etc., rely on Protected Mode's foundation.
- **Reverse Engineering**: You'll decode malware that plays with descriptor tables, hides in Ring 0, or abuses paging.
- **Rootkits**: Often mess with page tables or the GDT to hide processes, files, etc.

💡 Recap

- Real Mode = everyone in one room, no rules.
- Protected Mode = private rooms, locked doors, and security watching every hallway.
- Introduced with the 286, but real juice came with the 386 — 32-bit addressing, paging, multitasking, privilege levels.
- Without this? No Windows, no Linux, no apps sandboxed from each other.

Holy Rings of CPU Authority

Welcome to the **Holy Rings of CPU Authority**, where every line of code knows its place or gets slapped with a **General Protection Fault**.

This right here? The **Ring Model**? It's *the backbone* of modern computing security. It's how the CPU says:

"Respect the boundaries... or die screaming in exception #13."

Let's **revamp this section**, top-to-bottom, in true Gen Z-meets-binary-style.

🛡️ CPU Ring Levels (Privilege Levels 0–3)

"Bow before the Ring 0 throne, peasant app."

Think of the CPU as a medieval kingdom — with **Ring 0** as the king's throne room and **Ring 3** as the peasant village on the edge of the forest. The closer to Ring 0 you get, the more **power**, **trust**, and **danger** you wield.

🧩 What Are Rings?

Ring	Name	Who Lives Here	Power Level
0	Kernel Mode	OS kernel, drivers, system calls	🟢 All Access God Mode
1	Middle Privilege	Rare legacy stuff	🟡 Barely used
2	Same as above	Maybe firmware	🟡 Forgotten
3	User Mode	Apps, games, browsers	🔴 Restricted, sandboxed

🔥 Ring 0 — The Throne Room (Kernel Mode)

The code that runs here can:

- Access **any** memory
- Talk **directly** to hardware
- Execute all privileged CPU instructions (like modifying CR0, setting page tables, disabling interrupts)
- Literally reboot the system if it feels like it… 💀

👑 Who runs here?

- The **OS kernel** (Linux, Windows NT, macOS kernel, etc.)
- Kernel-mode **device drivers** (e.g., GPU drivers, disk I/O systems)
- Syscalls, trap handlers, and interrupt handlers

This is **where the CPU trusts you not to screw up**, because if you do... the whole system crashes. Not just your app — *the whole OS dies.*

📦 Ring 3 — The Sandbox (User Mode)

This is where:

- Your browser lives
- Your games run
- Your hacking tools load
- Your shady downloaded .exe lives

But the hardware *doesn't trust* this code.

Rules of Ring 3:

- Can't access **kernel memory.**
- Can't execute **privileged instructions** (cli, hlt, lgdt, etc.)
- Can't touch **I/O ports** or devices directly.
- Can't modify **page tables.**
- If you try? **General Protection Fault (#GP)** hits you like: *"Try that again and I'm calling the OS."*

🧠 Anything powerful must be requested via a **system call (syscall)** — that's Ring 3 politely knocking on Ring 0's door like:

```
read(fileDescriptor, buffer, size); // Hey kernel, can I access the file?
```

Ring 0 will check permissions, do the operation, then return control back to Ring 3. You don't get to touch the disk driver yourself.

🪦 Rings 1 and 2 — Ghost Towns of Privilege

Technically they exist, and the CPU will support them, but...

📉 **Modern OSes don't use them**.

- Linux and Windows both operate on a **two-ring model**:
  - Ring 0 = Kernel
  - Ring 3 = User

Why skip 1 and 2?

- Simplicity
- Easier OS design
- Already have privilege separation via **user/kernel and paging**

They were intended for things like:

- **Device drivers.**
- **Hypervisors.**
- **Firmware-level code.**

But modern systems moved those into:

- Ring 0 directly (for kernel-mode drivers)
- Or **VMX root mode** (for virtualization)
- Or **System Management Mode (SMM)** for firmware (deeper than Ring 0!)

📍 Why the Ring Model Matters

Feature	Benefit
🛡️ Security	Prevents user apps from crashing or hijacking the OS
🤝 Controlled Access	All critical hardware access must go through system calls
🚫 Isolation	One app crashing doesn't crash the kernel
🤓 Memory Protection	Ring 3 code cannot access Ring 0 memory space
📶 Fault Detection	Illegal access = immediate CPU trap = OS gets to respond

It's like turning the CPU from a lawless town into a **secure skyscraper with keycards**.

🧠 Real Example

Let's say a user-mode app tries to write directly to a hardware I/O port:

```
out 0x3F8, al  ; Try to send to COM1
```

👮 **CPU:**

"You're Ring 3. This is privileged. Not today."

💥 **#GP Fault**, control handed over to the OS.

💥 The process may be killed or sandboxed.

📌 Deep Tech Bonus — Paging + Rings = 🔐 Lockdown

Even with Ring 3 access, an app still lives in a **virtual memory space** defined by **page tables** owned by the kernel.

The CPU uses both:

- Ring level checks
- Page-level access flags (U/S, R/W, Present bits in page tables)

to fully enforce:

"This memory range belongs to the OS. You, little app, shall not pass."

🔚 Final Analogy: The Secure Castle

Ring 0: Royal Family & Guards (full access).

Ring 1–2: Knights & Messengers (mostly missing today).

Ring 3: Outsiders — must ask politely to enter, and only through the front gate.

Page Tables: Room access restrictions = Room Access Rules for Each Program.

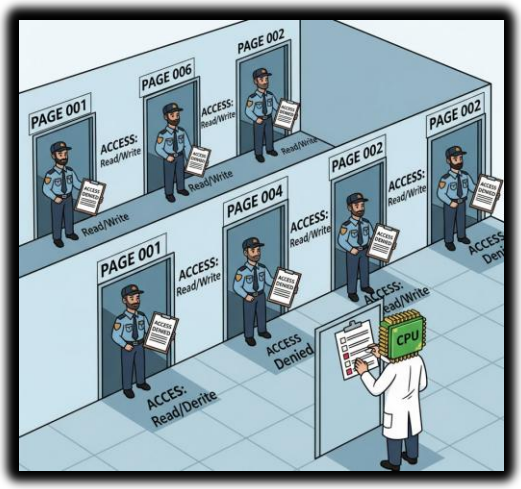
🔐 A page table is like a **custom rulebook (or access list) for one specific program's memory space**.

Each guest (program/process) gets their own keycard system (page table). The keycard only opens their own set of rooms — their virtual memory.

They can't open anyone else's rooms. Even if the rooms are physically close, the keycard doesn't allow it.

The front desk (Memory Management Unit, MMU) checks every room access using the guest's personal keycard (their page table).

Some rooms are read-only, others are read-write. Some may not even exist in memory yet (paging/swapping).



Control Registers = Castle's Blueprint + Secret Trapdoors.

The CPU is the Kingdom's Brain. Control Registers (CR0–CR4, CR8) are like master scrolls or switches in the castle's war room.

They define how the kingdom behaves — is it in peace mode or battle mode? Is the castle protected with traps or wide open?

Some key trapdoor options:

- *PAE (Physical Address Extension) — Allows castle to access more than 4GB of land.*
- *SMEP / SMAP — Stops even traitors (kernel bugs or malware) from accessing forbidden peasant zones (user-mode pages).*
- *PSE — Lets you open larger trapdoors (4MB pages instead of 4KB).*

This is the "We upgraded the castle" scroll. Most of this didn't exist back in the 80s.

General Protection Fault: Moat monster that eats intruders. An error that occurs when a program in memory tries to access an area of memory that it is not authorized to access

🛎️ Ring 3 ➜ Ring 0: How Syscalls Actually Work (Safely)

Okay, so apps in Ring 3 (user mode) are like:

"Yo Kernel, I need to read a file. Let me in for a sec?"

But they can't just JMP into Ring 0 code. That's illegal and would trigger a #GP fault. Instead, there's a whole trusted system for controlled privilege elevation — and it's 🔐 hardware enforced.

There are three main methods CPUs have used over the years to do this:

(this subtopic isn't as important for a beginner)

🧨 int 0x80 — Old School, Still Cool

This is the OG syscall method used in early Linux (and still supported today).

User code triggers a software interrupt:

```
mov eax, 1          ; syscall number for sys_exit
mov ebx, 0          ; exit code 0
int 0x80            ; interrupt vector 0x80, handled by the kernel
```

🧐 What happens:

- CPU does a **privilege check**.
- If int 0x80 is mapped to a **trap gate** in the IDT (Interrupt Descriptor Table), and the gate allows Ring 3 to call it (DPL=3)
- It **switches stacks** using ESP0 from the TSS.
- Jumps into Ring 0 and executes the syscall handler.

⚠️ Slow, but very **simple and secure**.
Used mostly on x86 (not x86_64), pre-SYSCALL era.

⚡ sysenter / sysexit — Fast Syscalls (Intel)

Introduced by Intel for speed. These don't use interrupt tables — they're special **fast-path CPU instructions**.

Set up first by the OS:

- ➢ MSRs (Model Specific Registers) are configured:
  - SYSENTER_CS_MSR → Ring 0 code segment
  - SYSENTER_ESP_MSR → Kernel stack pointer
  - SYSENTER_EIP_MSR → Address of syscall handler
- ➢ Then Ring 3 code does:

```
mov eax, syscall_number
sysenter
```

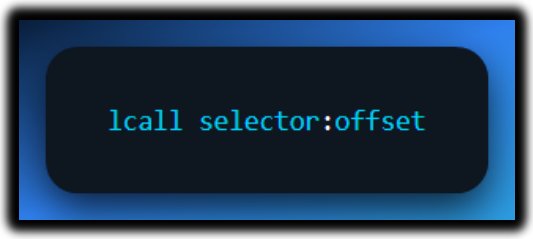
CPU switches instantly to Ring 0 using the values from MSRs. Way faster than int 0x80.

💡 But it's **one-way** — the sysexit instruction is used to return. More rigid than interrupts, but high-performance.

🧠 call gates — Fancy, Obsolete Mechanism

This is the textbook "Ring switching" method using **segmentation**:

- A descriptor in the **GDT** or **LDT** (check the 003 docx) defines a **Call Gate**:
  - ➢ It Includes:
    - Target segment (Ring 0)
    - Target offset (function entry)
    - DPL (who's allowed to call this)
    - Argument count
- User-mode code uses a **far CALL**:



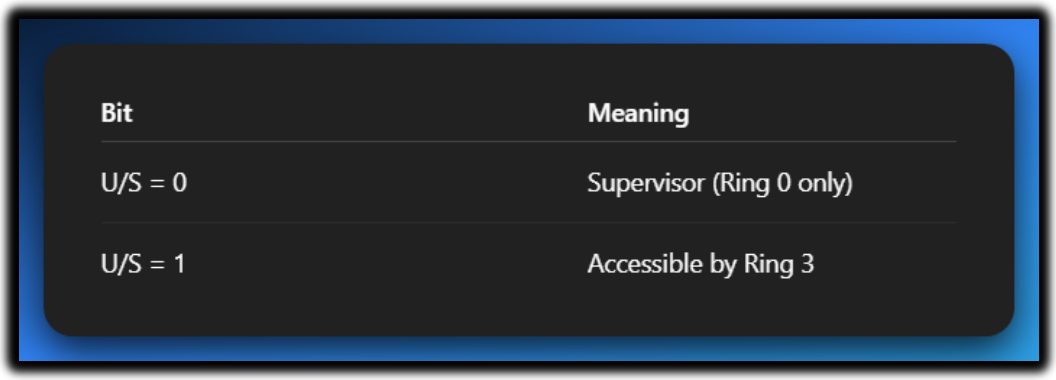
- CPU checks DPL & privilege, **switches stack**, loads new CS:IP, and runs the code in Ring 0.

👴 Nobody uses this now. Too slow and complex. But the mechanism is beautiful if you're into low-level syscall theory.

🔐 Paging Enforcement: The U/S Bit

Even **if** Ring 3 code tried to cheat via memory, it couldn't.

That's because in paging (via **page tables**), every page has a **U/S (User/Supervisor)** bit:



Bit	Meaning
U/S = 0	Supervisor (Ring 0 only)
U/S = 1	Accessible by Ring 3

So even if a Ring 3 app gets crafty and tries to:

```
void (*fun)() = (void*)0xC0000000;
fun();
```

💥 CPU checks page's U/S bit → it's 0 → access denied → **#PF** or **#GP** → app gets spanked.

void (*fun)(): *This declares a function pointer called fun that points to a function taking no parameters and returning void (nothing).*

(void*)0xC0000000: *This is the tricky part! It's casting the address 0xC0000000 (which is typically a high address, often in kernel or protected memory regions) to a void* (a pointer to anything).*

fun(); *This is calling the function through the pointer fun. But since fun is pointing to 0xC0000000, it's basically trying to execute whatever is at that memory address.*

OS: *"He's trying to execute random code from an inaccessible memory address, which is basically begging for chaos, let me call VMP to yoink this program to space!* 😤

Virtual Memory Protection: *Roger that! I see him...Activating yoink and yeet!*

Method	Used by	Speed	Notes
`int 0x80`	Linux x86	🐢 Slow	Software interrupt, easy
`sysenter/sysexit`	Intel CPUs	⚡ Fast	Needs MSR setup
`call gate`	Legacy OS	🪦 Obsolete	GDT-based Ring switch
Paging U/S bit	CPU/MMU	🛡️ Passive	Enforces privilege at memory level

😤 So yeah...

You're **in Ring 3**, asking nicely via syscall pathways like:

"Hey Kernel, can you open this file for me? I swear I won't segfault..."

And the CPU is the bouncer at the door like:

"Only if you come through the correct syscall handler. Otherwise... #GP fault. Back to userland."

FUN STUFF:

You can look at a **real Linux syscall implementation in C + ASM next**, or maybe break down the **IDT/trap gate setup for int 0x80**?

🧠 Final Impact of Protected Mode + Ring Model

Protected Mode isn't just a cool CPU mode — it's the **foundation of modern OS behavior**, **security**, and **performance**. Here's how it hits different:

🔐 1. System Security & Stability (Thanks to Rings)

🧱 The ring protection model is the **first line of defense** in modern computing.

- Malware running in **Ring 3** (userland) can't touch Ring 0 kernel code or memory directly.
- Any attempt to:
  - ➢ Write to kernel space.
  - ➢ Execute privileged instructions.
  - ➢ Hijack hardware I/O.
    triggers an **instant General Protection Fault (#GP)** or **Page Fault (#PF)** — CPU says **"NOPE."**

✅ If the OS is properly designed, this means:

- No app can corrupt another app's memory
- No rogue code can rewrite kernel space
- Malware is **sandboxed hard**

📉 If the ring model didn't exist? One crashy Chrome tab could take your whole OS down with it.

🧠 2. True Multitasking (With Memory Protection)

Protected Mode enables **context switching** — the OS can:

- Give each process a **time slice** on the CPU
- Save/restore register state, segment state, stack pointers
- Use **TSS** (Task State Segment) or software switching
- Keep each program isolated in its own memory space via **paging**

This allows:

- Multiple apps to run "simultaneously"

- Each one to think it owns the whole machine
- No app to corrupt another by accident (or on purpose)

It's the OS playing **juggler**, and Protected Mode gives it gloves and padding.

💯 3. Full Instruction Set Unlocked

In Real Mode, you're stuck with:

- 1MB address space
- 16-bit segments
- Limited access to instructions and no protection

But in Protected Mode:

- 🔓 You unlock **full 32-bit instruction set**
- Use **floating point math**, **SSE**, **MMX**, etc.
- Access full 4GB virtual address space
- Paging, segment limits, trap gates — all enabled

No more baby steps. You're in full CPU architect mode now.

🌎 4. Real-World Usage

Protected Mode isn't just theory — it's the **default mode** of operation for:

- All **32-bit operating systems** (Windows XP, Linux distros, DOS extenders like DPMI, etc.)
- **64-bit OSes running in compatibility mode** (like WOW32)
- Virtual machines that emulate x86 environments
- System-level tools, debuggers, and OS devs

Protected Mode is where the **real work gets done**, even today — especially for reverse engineers, malware analysts, and kernel developers who still mess with x86 internals.

🧠 TL;DR — Why You Should Care

3. Virtual-8086 Mode (VM86 Mode): Bridging the Past and Present

Virtual-8086 Mode is a special sub-mode of **Protected Mode**, introduced with the 80386 processor. Its purpose is to allow legacy 16-bit Real Mode applications (like old MS-DOS programs) to run within a modern multitasking Protected Mode environment without requiring any modifications to the old code.

- **The Problem it Solved:** Before VM86 mode, running a DOS program on a Protected Mode OS was problematic because DOS programs expected direct hardware access and had no concept of memory protection.
- **How it Works (Virtualization):**
    - The operating system, running in Protected Mode (Ring 0), creates a **virtualized 8086 environment** for each DOS program.
    - Each virtual-8086 "monitor" (part of the OS) intercepts privileged instructions and hardware access attempts made by the 16-bit DOS application.
    - Instead of letting the DOS program directly access hardware or memory (which would violate Protected Mode's rules and crash the system), the OS's virtual-8086 monitor **emulates** or **filters** these accesses. For example, if a DOS program tries to directly write to video memory, the OS intercepts this, redraws the virtual screen, and then updates the actual physical screen, giving each DOS program its own "virtual" display.
    - This provides a safe, isolated environment. If a DOS program crashes within its virtual-8086 session, it typically does not affect other programs or the stability of the main operating system.
- **Multitasking Legacy Apps:** A modern OS can run multiple, completely separate virtual-8086 sessions concurrently, allowing you to multitask several old DOS applications alongside your modern Windows or Linux applications.
- **Analogy:** Imagine a historical reenactment. The modern city (Protected Mode OS) sets up a special, fenced-off area (Virtual-8086 mode) where actors (DOS programs) can live and behave exactly as they did in the 1980s, completely unaware of the modern city around them. Any requests they make (e.g., for supplies) are handled by designated "handlers" (the OS's VM86 monitor) who translate their old-fashioned requests into modern actions, ensuring they don't break character or disrupt the present-day city.

4. Long Mode: The 64-bit Frontier

Long Mode is the operating mode utilized by all **64-bit x86 processors (x86-64 or AMD64 architectures)**. It's the latest major evolution of the x86 architecture, designed to

break the 4GB memory barrier and enable significantly larger address spaces and register sets.

- **Memory Model:**
    - **64-bit Addressing:** Long Mode fundamentally changes the addressing scheme to 64-bits (though current implementations typically use 48-bit virtual and 52-bit physical addresses). This allows access to vastly larger amounts of memory – theoretically up to 16 Exabytes (EB) of virtual address space and 4 Petabytes (PB) of physical address space – far exceeding anything previously possible.
    - **Mandatory Paging:** In Long Mode, **paging is mandatory**. Segmentation (as it was used for memory protection in 32-bit Protected Mode) is largely "flat" and effectively disabled for memory protection, with paging taking over that role entirely.
- **Increased Processing Power & Registers:**
    - **64-bit Registers:** General-purpose registers (like RAX, RBX, RCX, RDX, etc.) are extended to 64 bits, allowing the CPU to process larger chunks of data in a single operation.
    - **More Registers:** Long Mode also introduces 8 additional general-purpose registers (R8-R15) and 8 additional SSE/AVX registers (XMM8-XMM15/YMM8-YMM15), significantly improving performance by reducing the need to constantly load/store data from memory.
- **Sub-modes:** Long Mode itself has two primary sub-modes:
    - **64-bit Mode:** This is where true 64-bit applications and operating system kernels run, leveraging all the 64-bit features.
    - **Compatibility Mode:** This allows 32-bit and 16-bit Protected Mode applications to run on a 64-bit OS. The processor operates in a 32-bit or 16-bit environment within this mode, but the underlying OS is still in 64-bit Long Mode. **Crucially, Real Mode or Virtual-8086 Mode programs cannot be run natively in Long Mode; they typically require virtualization software (like a full VM) to run.**
- **Usage:** All modern 64-bit operating systems (Windows x64, Linux x64, macOS) and their applications run in Long Mode.

5. System Management Mode (SMM): The Covert Maintainer

System Management Mode (SMM) is a special-purpose, highly privileged operating mode intended for system-level functions, often transparent to the operating system and applications. It was introduced with the Intel 386SL.

- **Purpose:** SMM is primarily used by the **system firmware (BIOS/UEFI)** and hardware manufacturers for low-level system management tasks such as:
    - **Power Management (ACPI):** Adjusting CPU fan speeds, controlling power states (sleep, hibernate), battery management.
    - **System Security:** Handling specific security features (e.g., some forms of DRM, Trusted Platform Module interactions).
    - **Hardware Emulation/Compatibility:** Providing legacy support for certain hardware devices or handling specific chipset quirks. For instance, emulating a PS/2 mouse/keyboard for older OS components even if the physical devices are USB.
- **Entry Mechanism (SMI):** SMM is entered by a **System Management Interrupt (SMI#)** signal, which can be triggered by specific hardware events (e.g., pressing the power button, overheating) or by software writing to a special I/O port.
- **Privilege Level & Transparency:**
    - SMM code runs at an even *higher* privilege level than Ring 0 (kernel mode). It's sometimes informally called "Ring -2."
    - When an SMI occurs, the CPU saves its entire current state (registers, flags, etc.) into a special, protected memory area called **SMRAM (System Management RAM)**. The CPU then switches to SMM, executes the SMI handler code (located in SMRAM), and once the handler completes, it restores the CPU's original state from SMRAM and returns to the previous mode (e.g., Protected Mode).
    - This process is designed to be **completely transparent** to the operating system, which is unaware that the CPU temporarily left its control to perform SMM tasks.
- **Security Implications (for you!):** For reverse engineers and malware analysts, SMM is a particularly interesting (and dangerous) area. Because it's transparent to the OS and runs at such a high privilege, malware that manages to gain control within SMM can be incredibly difficult to detect, analyze, and remove. It can persist even across OS reboots and bypass many standard security measures. This makes SMM a prime target for sophisticated rootkits and bootkits.

Switching Between Modes: A Carefully Orchestrated Dance

Transitioning between these modes is not a simple flip of a switch. The processor must execute a specific instruction or a precise sequence of instructions to change modes. These transitions often require special privileges (usually only the operating system kernel can

initiate them) and involve careful manipulation of control registers (like CR0, CR4) and descriptor tables that define memory segments and pages.

- For instance, switching from Real Mode to Protected Mode involves enabling the "Protection Enable" (PE) bit in the CR0 control register and setting up segment descriptors and potentially page tables.
- Similarly, entering Long Mode requires first setting up a minimal Protected Mode environment, then enabling PAE (Physical Address Extension) in CR4, loading page tables, and finally setting the Long Mode Enable (LME) bit in a Model Specific Register (MSR) before enabling paging.

The OS as the Mode Manager

Ultimately, the **Operating System (OS)** is the arbiter of which mode the x86 processor operates in. Modern OSs start in Real Mode, then quickly transition to Protected Mode (for 32-bit OSs) or Long Mode (for 64-bit OSs) and stay there for the vast majority of their operation. Applications run within the context established by the OS, conforming to the rules and limitations of that mode (e.g., user-mode applications running in Ring 3).

Each mode defines its own distinct set of rules for:

- **Registers:** While many registers are consistent, their effective sizes and interpretations can change (e.g., 16-bit vs. 32-bit vs. 64-bit registers).
- **Memory Management System:** How addresses are translated from logical to physical, and how memory protection is enforced.
- **Interrupt Handling Mechanism:** How the CPU responds to hardware and software interrupts.

Understanding these modes is foundational. As you delve deeper into reverse engineering and malware analysis, you'll find that much of the challenge lies in understanding *which mode* code is executing in, how it got there, and what privileges it has based on that mode. Malware often tries to elevate its privilege or hide its presence by exploiting vulnerabilities related to these mode transitions or by executing in highly privileged modes like SMM.