When a huge application runs, the OS first **reserves virtual memory** for it — this just blocks out address space without using real RAM.

As the app starts executing, the OS **commits** the memory it actively needs — meaning it backs those virtual pages with **physical RAM** or **pagefile space**.

Whenever the app **accesses a new memory page**, the OS **commits** it and gives it real RAM.

If a committed page becomes **idle** (not accessed for a while), the OS may **swap it out**: it writes the page's data to the **pagefile** (a hidden file on disk) and **frees the physical RAM**.

Later, if the app accesses that swapped-out page, the CPU triggers a **page fault**, and the OS reads the data back from the **pagefile** into RAM and resumes execution.

Category	Stack	Heap
Definition	Fixed-size memory used for function calls and local variables	Dynamic memory allocated during runtime, explicitly managed
Allocation Method	Implicit (compiler handles it)	Explicit (`malloc`, `new`, `VirtualAlloc`, etc.)
Deallocation Method	Automatic (when function returns)	Manual (`free`, `delete`, `VirtualFree`)
Access Speed	Very fast (LIFO, pointer arithmetic only)	Slower (allocator overhead, fragmentation)
Lifetime of Allocations	Tied to function scope / call stack	Persists until manually freed or app exits
Growth Direction	Grows downward (from high addresses to low)	Grows upward (from low addresses to high)
Memory Size Limit	Small (usually 1–4 MB per thread, configurable)	Large (can be multiple GBs; limited by virtual memory commit space)
Managed By	Compiler and OS	Runtime allocator + OS
Thread Scope	Each thread gets its own stack	Shared between all threads unless explicitly isolated
Use Case Examples	- Local variables - Function params	- Objects with long lifetimes - Dynamic arrays

🧠 Key Takeaways

- **Stack** = short-lived, fast, local, safe, but **limited**
- **Heap** = long-lived, flexible, powerful, but **manual and riskier**
- Stack is **per-thread**, heap is **shared**
- Stack can never leak; **heap will absolutely leak** if you're careless
- Stack overflows = instant crash
  Heap misuse = silent corruption, leaks, eventual death

🧱 **Process** — Formal Definition

A **process** is:

> An instance of a running program that owns a private **virtual address space** and **system resources**, and acts as a container for one or more **threads** of execution.

🔍 It Contains:

- Virtual memory (reserved + committed pages)
- Executable code (loaded PE modules)
- Global/static variables
- Open handles (files, sockets, registry, mutexes)
- Environment block
- Security context (token, SID, privileges)
- One or more **threads**

🧠 Key Point:

A process **does not run on the CPU directly** — **threads do**.

🔧 **Thread** — Formal Definition

A **thread** is:

> The smallest unit of execution that the OS can schedule. It exists within a process and shares that process's resources, but has its own **stack**, **CPU context**, and **execution flow**.

🔍 It Has:

- Program counter (instruction pointer)
- CPU registers (RAX, RIP, RSP, etc.)
- Stack (local variables, return addresses)
- Thread-local storage (TLS)
- Thread ID (TID)
- State (ready, running, waiting, terminated)

🧠 Key Point:

A thread is what the **CPU actually runs**.
It executes code **within the context of a process**.

🧠 Memory Model (Thread vs Process)

Area	Process-wide?	Thread-specific?
Heap	✅ Shared	❌ Not separate
Global/static vars	✅ Shared	❌ Shared
Stack	❌ One per thread	✅ Unique per thread
Registers, PC	❌ Separate	✅ Per-thread
Handles	✅ Shared	❌ Not unique per thread unless duplicated manually

User mode and kernel mode are two distinct CPU privilege levels that define how much control code has over the system.

User mode is the restricted environment where all regular applications (browsers, games, JavaScript engines, etc.) run. Code in user mode can't directly access hardware, device drivers, or memory outside its own virtual address space. It operates in **Ring 3** on x86 architecture, meaning it's the least privileged level. If a user-mode application needs to perform a privileged operation (like reading a file or allocating memory), it must **ask the kernel** via a **system call** — the OS mediates access.

Kernel mode, on the other hand, runs at **Ring 0**, the highest privilege level. This is where the operating system core (like `ntoskrnl.exe` on Windows), device drivers, and low-level system services execute. Code running in kernel mode has **unrestricted access** to all hardware, system memory, and I/O. It can do anything: kill processes, write to any page, talk to the GPU, and even crash the entire OS if buggy.

The separation between these two modes is what **prevents apps from destroying the system**. It forms the backbone of OS security and stability. **JS, Python, C++, Java — they all run in user mode.** If they want kernel-level control, they either go through system APIs or load a **driver**, which is kernel-mode code.

In short: **user mode is the sandbox; kernel mode is the machine room**.

🧠 What Are CPU Rings?

CPU **rings** are **hardware-enforced privilege levels** used by processors (mainly x86/x64) to control **access to critical resources**.
They define how trusted a piece of code is, and what it's allowed to do.

Intel defines 4 rings:

Ring	Name	Privilege	Typical Use
0	Kernel Mode	🟢 Most privileged	OS kernel, drivers
1	Rare / Middle Layer	🟡 Rarely used	Hypervisors, OS subsystems (not common)
2	Rare / Middle Layer	🟡 Rarely used	Same as above
3	User Mode	🔴 Least privileged	Apps, games, browsers, JS, C++, Python, etc.

In practice, **only Ring 0 and Ring 3 are used.**

🔥 The Real Deal: Ring 0 vs Ring 3

Feature	Ring 0 (Kernel Mode)	Ring 3 (User Mode)
Can access hardware?	✅ Direct access	❌ Must use syscalls
Can access all memory?	✅ Full physical + virtual memory	❌ Only own virtual space
Can run privileged CPU instructions?	✅ Yes	❌ Will cause fault
Can load drivers?	✅ Yes	❌ No
Can crash the OS?	✅ Yes (BSOD risk)	❌ Only crashes itself

📦 Real Flow: Keystroke Example

Let's say your app wants to detect keystrokes via `GetAsyncKeyState()` or GLFW's key callback.

Here's the truth:

1. Your app runs in **Ring 3 (user mode)** — no hardware access.
2. You call a function like `GetAsyncKeyState(VK_SPACE)` → Win32 API.
3. This internally uses a syscall to ask the kernel → `"Hey kernel, what's the state of the keyboard?"`
4. The kernel (Ring 0) goes → `"Let me check the input buffer from the HID driver"`
5. Kernel responds → result sent back to user mode

Your app **never touched the hardware** — it just made an API call, which wrapped a syscall, which the kernel fulfilled.