## Part 1: The Basics of Memory & Addresses

Before a program can run, it has to be moved from the hard disk (where it's stored) into **main memory (RAM)**. That's because the CPU can only work with data that's in memory.

But the CPU doesn't access memory in just one way—it uses two main types of storage:

- **Registers** → Tiny storage inside the CPU itself. Extremely fast (almost instant).
- **Main Memory (RAM)** → Much larger, but slower.

To make things faster, computers place something called **cache** between the CPU and main memory. It acts like a shortcut for frequently used data.

Imagine multiple programs running at the same time. Without protection, one program could accidentally overwrite another program's data—which would crash everything. To prevent this, the system uses two special registers:

- **Base Register** → Holds the starting address of a program in memory
- **Limit Register** → Holds the size (length) of that program

### Address Binding

When a program runs, its instructions need memory addresses.
**Address binding** is simply the process of assigning those addresses. This can happen at three different times:

### 1. Compile Time (Before running)

- Done by the programmer/compiler
- If you already know where the program will be placed in memory, the addresses are fixed ("hardcoded")
- Problem: If that location is unavailable later, you must recompile the program

### 2. Load Time (When the program starts)

- Done by the operating system
- The OS chooses a free spot in memory when loading the program
- Addresses are fixed at that moment

### 3. Execution Time (While the program is running)

- Done by both hardware and the OS
- The program can be moved around in memory while running
- Addresses must be updated dynamically– this is why it is what modern systems use

## Logical VS. Physical Addresses

To understand execution-time binding, we need to distinguish between two types of addresses. Think of it like this: Your program is given a *fake map* (logical addresses), and the system secretly translates it into the *real map* (physical addresses).

- **Logical (Virtual) Address**
    - Generated by the CPU
    - What the program *thinks* it is using
    - Usually starts from 0
    - Not real—it's an abstraction
- **Physical Address**
    - The actual location in RAM
    - Used by the hardware

## The MMU: The Translator

Since programs use logical addresses, something must convert them into real ones. That's the job of the **Memory Management Unit (MMU)**, the translation happens instantly while the program runs. The **MMU** exists because of **execution-time binding**.

Here's how it works:

1. **Check Safety (Limit Register)**
    - The MMU first checks if the logical address is within the program's allowed size
    - This prevents accessing memory that doesn't belong to the program
2. **Translate Address (Base Register)**
    - The base register holds the real starting location of the program
    - The MMU adds:
      **physical address = base address + logical address**

## Old Systems (No MMU)

Programs stayed in one fixed place in memory because addresses were always physical and hardcoded. No translation was needed… But this was inefficient—programs couldn't move.

## Modern Systems (With MMU)

Programs can move around in memory while running, CPU generates logical addresses while the MMU translates them into physical addresses. The base and limit registers provide safety and flexibility.

## Part 2: Dynamic Loading, Linking, and Swapping

When a program runs, it must first be brought from secondary storage (such as a hard disk or SSD) into main memory (RAM), because the CPU can only execute instructions that reside in memory. However, memory is limited, and modern systems often run multiple programs at the same time. For this reason, operating systems use several techniques to load programs efficiently and to manage situations when memory becomes full. The most important of these techniques are **dynamic loading**, **dynamic linking**, and **swapping**.

## Dynamic Loading and Dynamic Linking

One of the key observations in program execution is that not all parts of a program are used all the time. Many programs contain optional features, error-handling routines, or rarely used functions that may never actually execute during a typical run. Loading the entire program into memory at once would therefore waste valuable space.

Dynamic loading solves this problem by loading routines into memory **only when they are needed**. Instead of placing every function in memory at the start, the operating system delays loading certain parts of the program until they are explicitly called during execution.

This process relies on a small piece of code known as a **stub**. A stub acts as a placeholder for a routine that has not yet been loaded. When the program attempts to call that routine, the stub takes control. It locates the required code on disk (or in a shared library), loads it into memory, and then transfers execution to it. Once loaded, the routine behaves like any other part of the program.

In dynamic loading, the stub **does not replace "code"** in a vague sense, but rather participates in replacing an **address reference** with the correct one at runtime. When a program is first loaded, calls to certain routines do not point to the actual function code. Instead, they point to the stub. This means:

- The program's instruction (for example, a function call) initially contains the **address of the stub**, not the real routine.
- When this instruction executes, control jumps to the stub.

At this point, the stub performs the following steps:

1. It locates the required routine in secondary storage (or a shared library).
2. It loads that routine into main memory.
3. It then **modifies the program's address reference**—specifically, it replaces the stored address of the stub with the **actual memory address of the loaded routine**.
4. Finally, it transfers execution to the real routine.

Thus, the stub acts as a **temporary address placeholder**, not a permanent piece of logic.

Closely related to dynamic loading is **dynamic linking**, which applies the same idea to libraries. Instead of linking libraries to a program before execution, the system postpones this step until runtime. This allows multiple programs to share a single copy of a library in memory, rather than each program having its own separate copy.

The benefits of dynamic loading and linking include:

- Reduced memory usage, since unused code is never loaded
- Faster program startup time
- Efficient sharing of common libraries among multiple programs

## Swapping

Even with efficient loading techniques, main memory can still become full when many processes are running simultaneously. To handle this, the operating system uses a technique called **swapping**.

Swapping allows the OS to temporarily remove a process from main memory and store it on secondary storage, known as the **backing store**. The backing store is typically a fast disk or SSD that is large enough to hold copies of processes currently not in memory. When memory is needed for another process, the OS selects a process (often a lower-priority one) and **swaps it out** to the backing store. Later, when that process needs to resume execution, it is **swapped back in** to main memory.

A common variation of this technique is known as **roll out, roll in**, which is used in priority-based scheduling. In this case:

- A lower-priority process is rolled out (removed from memory)
- A higher-priority process is immediately loaded and executed

The term **bottleneck** refers to any part of a system that limits overall performance due to being significantly slower than other components.

In the context of memory systems:

- The CPU can execute instructions extremely quickly (in nanoseconds).
- Main memory (RAM) is slower.
- Secondary storage (disk or SSD) is much slower.

Because of this mismatch, the CPU often has to **wait idle** for data to arrive from memory. This waiting time is the **memory bottleneck**.

The name comes from the shape of a bottle: A wide base (fast CPU capability) , A narrow neck (slow memory access)

No matter how fast the base is, everything must pass through the narrow neck, limiting throughput. This is precisely why mechanisms like **cache** and **dynamic loading** are important—they reduce how often the system must access slower memory layers.

And so while swapping provides flexibility, it has a significant drawback: it is slow. The time required to swap a process depends largely on how much data must be transferred between memory and disk. Since disk access is much slower than memory access, frequent swapping can severely degrade system performance. For this reason, modern operating systems such as Windows, Linux, and UNIX use optimized or partial forms of swapping, often combined with more advanced techniques like paging.

**Memory Management in Mobile Systems**

Mobile devices, such as smartphones and tablets, handle memory management differently from traditional desktop systems. Standard swapping is generally avoided on mobile platforms due to the limitations of flash storage. Flash memory gets its name from how it erases data. Unlike traditional memory, which erases data gradually or byte-by-byte, flash memory erases entire blocks of data **in a single, fast operation**, similar to a camera flash. Early engineers described this as happening “in a flash,” and the name remained.

Flash memory has several constraints:

- It has a limited number of write cycles before it begins to degrade (this happens because writing data involves **forcing electrons through an insulating layer.** Over time, this damages the material, making it unreliable)
- It offers lower performance for frequent large data transfers compared to RAM
- It typically has limited storage capacity

## Flash VS Disk (SSD)

At first glance, SSDs (Solid State Drives) appear not to suffer from this limitation, but in reality, they **use the same flash memory technology** and do have write limits.

The difference is that SSDs use advanced techniques to hide and manage this limitation:

- **Wear leveling:** Spreads writes evenly across all cells to prevent early failure of specific areas
- **Over-provisioning:** Includes extra hidden memory cells to replace worn-out ones
- **Error correction codes (ECC):** Detects and fixes small data errors

Even though SSDs appear durable and long-lasting from a user’s perspective, even though the underlying limitation still exists.

## iOS vs Android Memory Management

Although both iOS and Android avoid traditional swapping, they differ significantly in how they manage memory under pressure.

In **iOS**, memory management is handled in a proactive and controlled manner. When the system detects low available memory, it first requests applications to voluntarily reduce their memory usage. This means that apps are expected to actively release any memory they no longer need.

For example:

- Read-only data or cached content may be discarded from memory
- This data can later be reloaded from flash storage if required

This approach relies heavily on cooperation from applications. However, if an application fails to release enough memory in response to system warnings, the operating system may forcibly terminate it to maintain system stability. This design ensures that iOS maintains smooth performance, but it also places strict responsibility on applications to manage their own memory usage efficiently.

**Android** uses a more automatic and aggressive strategy compared to iOS. When available memory becomes low, the system prioritizes keeping critical applications running while terminating less important ones.

Before an application is terminated, Android performs an important step: it saves the current state of the application to flash storage. This includes information such as the user interface state and navigation history. As a result, when the user reopens the application, it can resume quickly from where it left off.

This approach allows Android to:

- Free memory quickly by removing background processes
- Preserve user experience by enabling fast restoration of applications

Unlike iOS, Android does not rely as heavily on applications voluntarily freeing memory. Instead, it manages memory more directly at the system level. Both systems rely on more efficient memory management techniques, such as paging, rather than full process swapping.

## Part 3: Memory Allocation Techniques

In early memory management systems, the operating system used a simple approach called **contiguous memory allocation**. In this method, main memory is divided into two main regions:

- One region stores the **operating system**, usually placed in low memory addresses
- The other region stores **user processes**, placed in high memory addresses

When a program is loaded from the hard disk into main memory, the operating system must find a **single continuous block of free memory large enough to hold the entire program**. This strict requirement is the core limitation of contiguous allocation: each process must occupy one uninterrupted region of memory.

## 1. Execution-Time Binding and the Role of the MMU

A crucial concept that enables modern memory management is **execution-time address binding**, supported by the **Memory Management Unit (MMU)**.

Execution-time binding allows a process to be:

- Assigned memory dynamically while it is running
- Moved between different physical memory locations during execution
- Kept unaware of its actual physical position in memory

This is possible because programs do not directly use physical addresses. Instead the CPU generates **logical (virtual) addresses** and the MMU translates them into **physical addresses at runtime.** This separation is what allows modern systems to introduce advanced memory management techniques such as **Contiguous memory allocation** (historical model), **Paging** (modern dominant model) and **Segmentation** (logical structuring model). Without execution-time binding, none of these techniques would be practical in dynamic systems, because programs would break whenever their physical location changed.

## 2. Hardware Protection Using Base and Limit Registers

Once a process is placed into memory, the system must ensure it does not access memory belonging to other processes or the operating system. This is achieved using two hardware registers:

- **Base Register:** Stores the lowest physical address assigned to the process
- **Limit Register:** Stores the size of the process (maximum addressable range)

Every memory access is checked as follows:

- The address must be **greater than or equal to the base register** and **less than base + limit**

The MMU performs this checking and translation dynamically during execution, ensuring memory safety and isolation between processes.

## 3. Finding Space: The Dynamic Storage Allocation Problem

As processes are loaded and terminated, memory becomes fragmented into scattered free regions known as **holes**. The operating system must decide where to place new processes

using a strategy called the **dynamic storage allocation problem**. Although these strategies differ in efficiency, they all suffer from the same fundamental issue: fragmentation over time.

Common strategies include:

- **First-fit:**
  The OS allocates the first hole large enough to fit the process. This is fast and commonly used.
- **Best-fit:**
  The OS searches all holes and selects the smallest one that fits, minimizing leftover space but increasing search time.
- **Worst-fit:**
  The OS places the process in the largest available hole, attempting to avoid small unusable fragments.

Fragmentation is an inherent problem in memory systems, but **contiguous memory allocation suffers the most**, which is why it is considered the oldest and least scalable memory organization technique.

It appears in two forms:

## External Fragmentation

As processes are loaded and removed, memory becomes split into many small, scattered free blocks. Even if there is enough total free memory, it cannot be used if it is not continuous.

This is called **external fragmentation**. A classical solution is **compaction**, where the operating system temporarily stops processes, moves them in memory and merges scattered free spaces into one large block

However, compaction is very slow, computationally expensive and requires moving large amounts of memory. Even though compaction solves external fragmentation, it is inefficient in practice. This is one of the key reasons modern systems prefer other approaches such as paging, which completely avoids external fragmentation instead of trying to fix it.

## Internal Fragmentation

Internal fragmentation occurs when memory is allocated in fixed-size blocks and the process does not fully use the assigned space. The unused portion remains inside the allocated block and cannot be used by other processes.

Unlike external fragmentation it occurs inside allocated memory and is unavoidable in fixed-size allocation systems. so there is no perfect solution to it. However, it can be **managed and reduced** using techniques like paging with appropriately chosen page sizes.

## 4. Why Execution-Time Binding Is Essential in Modern Systems

Execution-time binding is essential because modern operating systems must support **dynamic relocation of processes during execution**. This includes several operations:

- **Swapping (roll out / roll in):**
  Entire processes are moved between RAM and disk, often replacing low-priority processes with higher-priority ones.
- **Compaction:**
  Processes are moved to reorganize memory and reduce fragmentation.
- **Paging:**
  Parts of a process (pages) are moved independently between memory and disk.
- **General memory optimization:**
  The OS continuously rearranges memory to improve efficiency and allocation success rates.

All of these operations require that a process be movable without breaking its execution, which is only possible because of execution-time binding. Without it, any movement of a process would invalidate its memory references and cause system failure.

# Part 4: Paging

Paging is the modern memory management technique designed to eliminate the major weaknesses of contiguous memory allocation, especially **external fragmentation** and the need for compaction. Instead of forcing an entire program into one continuous block of memory, paging allows a program's memory to be stored **non-contiguously across main memory**. This flexibility is what makes paging one of the most important concepts in modern operating systems.

## 1. Basic Idea: Pages and Frames

Paging works by dividing memory into fixed-size blocks on both sides:

- The operating system divides **physical memory (RAM)** into fixed-size blocks called **frames**
- It divides a program's **logical memory** into blocks of the same size called **pages**

Because pages and frames are equal in size, any page can fit into any frame. So if a program consists of, for example, 4 pages: Those 4 pages can be placed in **any 4 free frames anywhere in memory.** This completely removes the requirement for contiguous memory. This is why paging eliminates **external fragmentation entirely**- so no **compaction** required, and it simplified allocation.

## 2. Logical Address Structure: (p, d)

When paging is used, the CPU no longer generates a simple physical address. Instead, it generates a **logical (virtual) address**, which is split into two parts:

- **p (Page Number):** Identifies which page of the program is being accessed
- **d (Offset):** Identifies the exact location inside that page

So instead of thinking in one linear address space, the CPU thinks in two levels:

> "Which page am I in?" + "Where inside that page?"

## 3. The Page Table: Mapping Logical to Physical Memory

Because a process's pages are scattered across physical memory, the operating system uses a **page table** to record where each page is stored.
Each process has its own page table in main memory. The CPU uses a special register called the **PTBR (Page Table Base Register)** that stores the address of the current process's page table.
During a **context switch**, the operating system loads the PTBR with the address of the next process's page table, which it obtains from that process's **PCB (Process Control Block)**. The Page Table acts as a **translation map between logical and physical memory**.

Paging works like Contiguous Memory in the sense that both convert a logical address into a physical one, but paging does it by first translating a page number into a frame, while contiguous memory directly adds a base register to an offset.

## Contiguous memory

- Logical address = offset
- Physical address = base + offset

## Paging

- Logical address = page number p , offset d
- Page number p → frame number d
- Physical address = frame base + offset d

Note: frame number must be converted into frame base address first and this is usually computed automatically by computer

> frame base = frame number × frame size

If frame/page size is given and this computation is NOT done, then:

Physical address = (frame number × frame size) + offset

## Important insight

The page table does NOT store actual data. It only stores **where each page is located in RAM.** So it is essentially a **directory or mapping system**, not memory itself.

## 4. The Performance Problem: Two Memory Accesses

Paging introduces a hidden cost. Since the **page table is stored in main memory**, every memory access requires:

1. One access to read the page table entry (to find the frame number)
2. A second access to read the actual data from that frame

So instead of one memory access, we now need **two memory accesses**. This effectively doubles memory access time, which is a major performance issue.

### 5. The Solution: TLB (Translation Lookaside Buffer)

To solve this performance problem, modern systems use a special high-speed hardware cache called the **TLB (Translation Lookaside Buffer)**. It is a very small, very fast memory cache **inside the CPU's Memory Management Unit (MMU)**. The TLB stores **recent page-to-frame translations**, so the system does not always need to access the full page table in main memory.

## How the TLB works:

When the CPU generates a logical address (p, d):

1. The system first checks the **TLB**
2. If the translation is found → **TLB hit**
    - The frame number is returned immediately
    - No need to access the page table in RAM
3. If the translation is not found → **TLB miss**
    - The system accesses the page table in main memory
    - The result is then stored in the TLB for future use

## 6. Trade-off: Internal Fragmentation Still Exists

While paging solves external fragmentation, it still has Internal Fragmentation. The last page of a process may not be fully used. This wasted space inside a frame is called **internal fragmentation**

However: It is predictable, usually small (depending on page size) and is far easier to manage than external fragmentation.

## Part 5: Segmentation

Segmentation is a memory-management scheme that organizes memory according to how a programmer naturally thinks about a program. Unlike paging, which divides memory into fixed-size blocks for hardware convenience, segmentation divides a program into **logical units that vary in size depending on their purpose**.

This makes segmentation a more “human-oriented” approach to memory management, because it matches the internal structure of software rather than the constraints of hardware.

## 1. The Core Idea: Logical Chunks of a Program

In segmentation, a program is not treated as one uniform block of memory. Instead, it is divided into meaningful components called **segments**, such as:

- A **code segment** (program instructions)
- A **data segment** (global variables and constants)
- A **stack segment** (function calls, local variables)
- A **heap segment** (dynamically allocated memory)
- Sometimes even individual modules or functions

Each of these segments:

- Has a **different size**
- Grows independently (especially stack and heap)
- Represents a logical part of the program

This is the key difference from paging: segmentation reflects *program structure*, not hardware structure.

## 2. How Segmentation Works: Segment Table and Address Translation

Because segments are variable in size and stored anywhere in physical memory, the system needs a structure to keep track of them. This is done using a **Segment Table**.

Each entry in the segment table contains two critical pieces of information:

**Base:** The **physical starting address** of the segment in main memory, tells the system *where the segment begins*

**Limit:** The **size of the segment,** defines how far the segment extends from the base

## 3. Logical Address in Segmentation

In segmentation, the CPU generates a logical address of the form:

(segment number, offset)

Where:

- **Segment number** identifies which logical part of the program is being accessed
- **Offset** specifies the position inside that segment

## 4. How Address Translation Works

When a memory access occurs:

1. The CPU provides a logical address: (segment number, offset)
2. The segment number is used to index the **segment table**
3. The system retrieves:
    - Base address of the segment
    - Limit (size of the segment)
4. The hardware performs a safety check:
    - It ensures that the offset is **less than the limit**
    - This prevents illegal memory access outside the segment
5. If valid, the final physical address is computed as:

Physical Address = Base + Offset

**Note:** Segmentation behaves very similarly to a **relocation register system**, but extended because segmentation allows **multiple base-limit pairs per process**, not just one.

- Base register → segment base
- Limit register → segment size
- Offset → position inside segment

## 5. Advantages and Disadvantages of Segmentation

Segmentation provides a very important advantage over paging: **semantic meaning**. Because memory is divided logically, the operating system can:

- Protect individual segments separately (e.g., make code read-only)
- Share specific segments between processes (e.g., shared libraries)
- Allow independent growth of stack and heap

This makes segmentation extremely useful from a **programmer's perspective**, because it aligns with how programs are actually written and structured.

**BUT:** Despite its logical elegance, segmentation suffers from a serious problem. Since segments are variable-sized, and must still be stored in contiguous memory regions. This leads to **external fragmentation**. Over time, as segments are allocated and removed, memory becomes divided into scattered free holes. Even if the total free memory is sufficient, it may not be usable if it is not contiguous.

To solve this, the operating system can perform **compaction**, However, we already discussed how slow and inefficient this solution is. This makes it impractical in modern high-performance systems. This is why **Segmentation Is Not Used Alone.** Because segmentation still suffers from external fragmentation, it is rarely used as a standalone system in modern operating systems.

Instead, modern systems prefer **paging**, but segmentation is still important conceptually and is sometimes combined with paging. For this reason, modern systems typically combine segmentation with paging, or rely primarily on paging alone. This hybrid approach allows systems to maintain logical structure while also achieving efficient, fragmentation-free memory management.

## 6. Real Hardware Example: Intel x86 (Segmentation + Paging Together)

One of the modern processors combine both segmentation and paging to get the benefits of each is the **Intel Pentium architecture**. The process works in two stages:

### Stage 1: Segmentation Unit

- The CPU generates a logical address
- The segmentation unit converts it into a **linear address**
- This step applies logical structure (segments)

### Stage 2: Paging Unit (MMU function)

- The linear address is then passed to the paging system
- The paging unit divides it into pages and frames
- It translates the linear address into a **physical address**

### Final result

> Segmentation provides logical organization, while paging provides physical efficiency.

So the system achieves:

- Programmer-friendly structure (segmentation)
- Hardware-efficient memory allocation (paging)