Chapter 15: Mechanism: Address Translation

This chapter introduces the fundamental mechanism that enables memory virtualization: **hardware-based address translation**. This technique is central to how an OS provides each process with the illusion of its own private memory, while maintaining efficiency and control.

> **The Crux of the Problem: How to Efficiently and Flexibly Virtualize Memory?**
>
> How can we build an efficient virtualization of memory? How do we provide the flexibility needed by applications? How do we maintain control over which memory locations an application can access? How do we do all of this efficiently?

The solution is a partnership between the OS and the hardware. The hardware provides the low-level mechanism for **address translation**, while the OS manages memory and sets up the hardware correctly.

1. The Core Idea: Address Translation

- **Concept:** The hardware, on every single memory access (instruction fetch, load, store), automatically transforms the **virtual address** generated by the program into a **physical address** where the data actually resides.
- **Interposition:** This is a powerful form of interposition. The hardware intercepts every memory reference and translates it, making the process transparent to the running program.
- **The Illusion:** The program operates as if it has its own private, contiguous memory space starting at address 0. The OS and hardware work behind the scenes to map this illusion onto the fragmented, shared reality of physical memory.

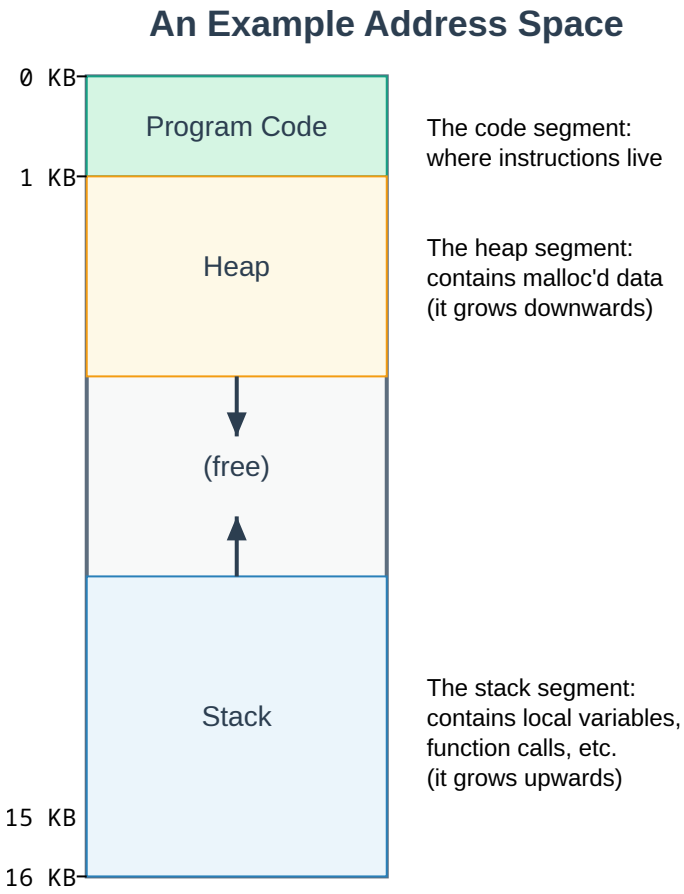
---

2. A Simple Example

Consider a program with a 16KB address space. Its code is at virtual address `128`, and a variable `x` is on the stack at virtual address `15KB`.

- **Program's View (Virtual Addresses):**
  - Fetch instruction from `128`.
  - Load data from `15KB`.
  - Store data to `15KB`.
- **System's View (Physical Addresses):**
  - The OS might load this entire 16KB address space into physical memory starting at address `32KB`.
  - The program's virtual addresses must be translated to physical addresses.
    - Virtual `128` -> Physical `32KB + 128`

* Virtual `15KB` -> Physical `32KB + 15KB`



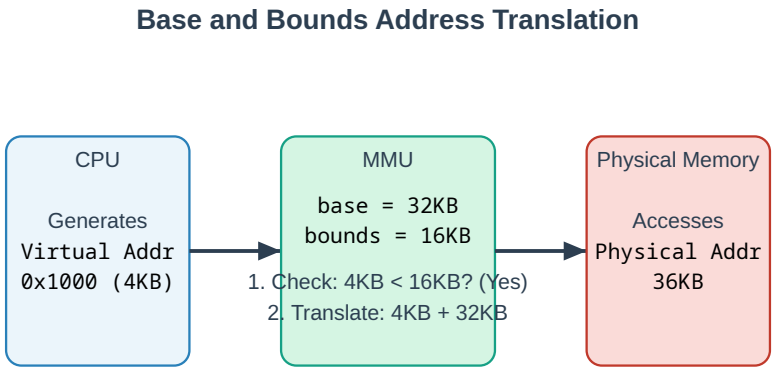
3. Mechanism: Base and Bounds (Dynamic Relocation)

The earliest hardware mechanism to support address translation is known as **base and bounds**, or **dynamic relocation**.

- **Hardware Requirement:** Two special registers per CPU, managed by the **Memory Management Unit (MMU)**:
  1. **Base Register:** Holds the starting *physical address* where the process's address space has been loaded.
  2. **Bounds (or Limit) Register:** Holds the *size* of the process's address space.
- **Translation Process:** For every virtual address generated by the program,

the hardware performs the following steps:

1. **Protection Check:** Is `virtual_address` within the allowed range? (i.e., `0 <= virtual_address < bounds`). If not, raise a hardware exception (a fault) and trap to the OS.
2. **Translation:** If the check passes, calculate the physical address: `physical_address = virtual_address + base`
3. The hardware then uses this physical address to access main memory.



4. Hardware and OS Responsibilities

Making this system work requires a clear division of labor between the hardware and the OS.

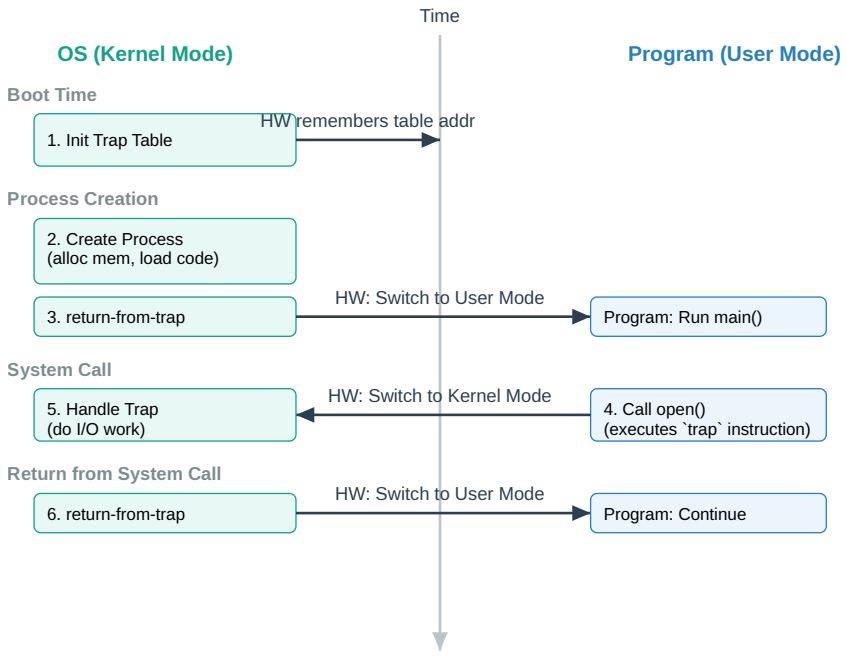
4.1. Hardware Responsibilities

- **CPU Modes:** Provide at least two modes: a privileged **kernel mode** for the OS and a restricted **user mode** for applications.
- **Base and Bounds Registers:** Provide the registers themselves.
- **Translation Circuitry:** The MMU must perform the `add` and `compare` operations for every memory reference, automatically and efficiently.
- **Privileged Instructions:** Provide instructions to modify the base and bounds registers. These instructions must only be executable in kernel mode.
- **Exception Handling:** The CPU must be able to raise exceptions (trap to the OS) when a program attempts an illegal action, such as:
  - Accessing memory outside its bounds.
  - Executing a privileged instruction in user mode.

4.2. Operating System Responsibilities

The OS manages the memory and configures the hardware to make virtualization work.

- **Memory Management:** When a process is created, the OS must find a free, contiguous chunk of physical memory large enough for its address space. It tracks free and used memory slots, often using a **free list**. When a process terminates, the OS reclaims its memory.
- **Context Switching:** The base and bounds registers are part of a process's context. When the OS performs a context switch:
  1. It **saves** the base and bounds values of the outgoing process to its Process Control Block (PCB).
  2. It **restores** the base and bounds values for the incoming process from its PCB into the hardware registers.
- **Exception Handling:** The OS must install handlers for memory-related exceptions at boot time. If a process causes a bounds violation, the handler will typically terminate the offending process.



5. Limitations of Base and Bounds

While efficient and protective, this simple dynamic relocation scheme has a significant drawback:

- **Internal Fragmentation:** A process's address space is composed of code, a heap, and a stack, with a large, unused "free" region in between. When the OS allocates a single contiguous chunk of physical memory for the entire address space, this unused internal region is also allocated, wasting physical memory. This waste is called **internal fragmentation**.

This inefficiency means we need more sophisticated mechanisms to better utilize physical memory. This leads to the concept of **segmentation**, which is the topic of the next chapter.