Chapter 16: Mechanism: Segmentation

This chapter introduces **segmentation**, an early and influential mechanism for memory virtualization. It generalizes the base-and-bounds approach to solve the problem of wasted physical memory caused by large, unused gaps within a process's address space.

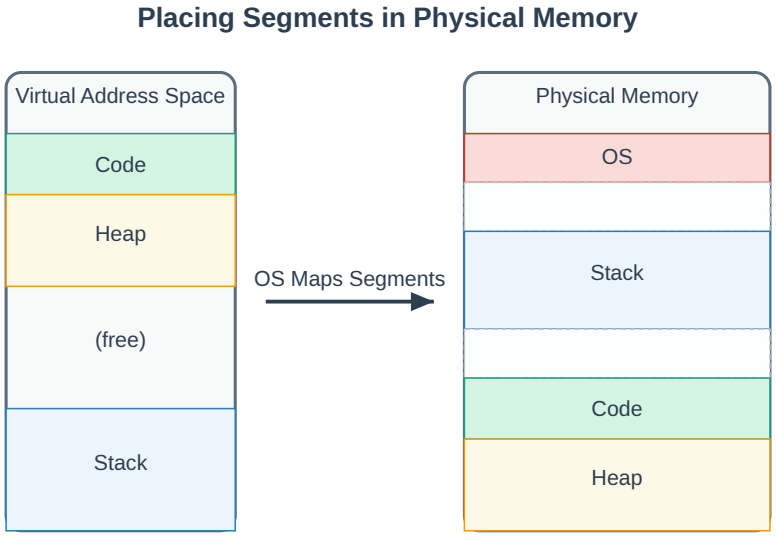
> **The Crux of the Problem: How to Support a Large Address Space?**
>
> How do we support a large address space with (potentially) a lot of free space between the stack and the heap without wasting physical memory?

Simple base-and-bounds requires allocating a single, contiguous block of physical memory for the *entire* virtual address space. This is wasteful, as the space between the heap and stack is allocated but unused. This is known as **internal fragmentation**.

1. Segmentation: Generalized Base and Bounds

The core idea of segmentation is to divide the address space into logical **segments** and give each segment its own pair of base and bounds registers.

- **Logical Segments:** A typical address space has three logical segments:
  1. **Code:** Contains the program's instructions.
  2. **Heap:** For dynamically allocated data.
  3. **Stack:** For local variables and function calls.
- **Mechanism:** Instead of one base/bounds pair, the Memory Management Unit (MMU) has a set of them—one for each segment. This allows the OS to place each segment independently in different, non-contiguous parts of physical memory.



This approach solves the internal fragmentation problem because the large, unused space between the heap and stack is no longer allocated in physical memory.

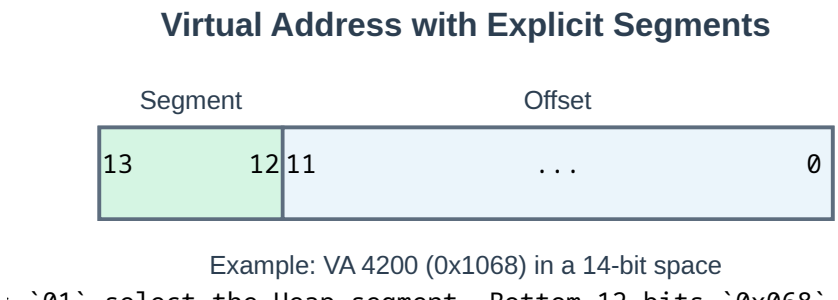
2. Address Translation with Segmentation

With multiple segments, the address translation process becomes slightly more complex. The hardware needs to know which segment a virtual address refers to.

2.1. Identifying the Segment

There are two main approaches for the hardware to determine the target segment:

- **Explicit Approach (Top Bits):** The top few bits of the virtual address are used to explicitly select the segment.
  - **Example:** In a 14-bit virtual address space with three segments, we could use the top 2 bits to identify the segment (e.g., `00` for code, `01` for heap, `11` for stack). The remaining 12 bits form the **offset** within that segment.



- **Implicit Approach:** The hardware determines the segment based on how the address was generated.
  - If the address comes from the Program Counter (instruction fetch), it's in the **code** segment.
  - If the address is based off the stack or base pointer, it's in the **stack** segment.
  - All other addresses are assumed to be in the **heap** segment.

2.2. The Translation Process

Once the segment is identified, the hardware performs the translation:

1. **Extract Offset:** The hardware calculates the offset of the virtual address within its segment.
   - For the code segment (starting at VA 0): `offset = virtual_address`
   - For the heap segment (starting at VA 4K): `offset = virtual_address - 4096`
2. **Check Bounds:** The hardware checks if the offset is within the segment's bounds (`0 <= offset < bounds`). If not, it raises a **segmentation fault**.
3. **Calculate Physical Address:** If the check passes, the hardware adds the offset to the segment's base register: `physical_address = segment_base + offset`.

2.3. Handling the Stack Segment

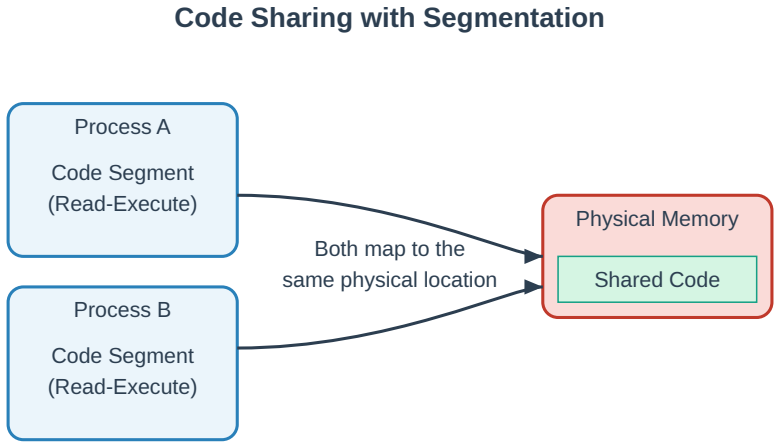
The stack grows in the opposite direction (downwards in virtual address space). This requires extra hardware support.

- **Growth Direction Bit:** The MMU uses an extra bit per segment to indicate its growth direction (e.g., 1 for positive, 0 for negative).
- **Translation:** For a negatively-growing segment, the offset is calculated differently. For example, `offset = virtual_address - max_segment_address`. This negative offset is then added to the base to get the physical address.

3. Support for Sharing

Segmentation naturally enables sharing of memory segments between processes, which is highly efficient.

- **Mechanism:** The hardware includes **protection bits** for each segment (e.g., Read, Write, Execute).
- **Code Sharing:** By setting a code segment's protection bits to **read-only** and **execute-only**, the OS can map the *same physical memory* containing the code into the virtual address spaces of multiple processes. Each process thinks it has its own private copy, but they are all safely sharing one physical copy, saving a great deal of memory.

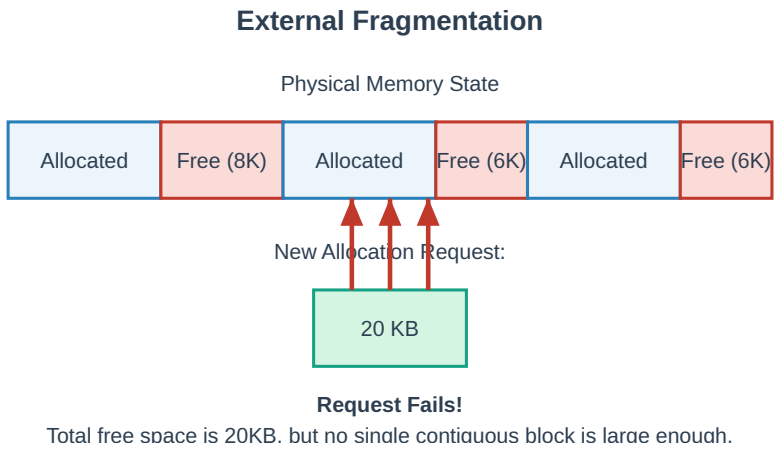


4. OS Support for Segmentation

The OS has several key responsibilities to manage a segmented memory system:

1. **Context Switching:** When switching between processes, the OS must save the segment registers of the old process and restore the segment registers for the new one.
2. **Managing Segment Growth:** If a process's heap or stack needs to grow beyond its current segment size, the memory allocation library will make a system call (e.g., `sbrk()`). The OS must then find physical memory to accommodate the growth and update the segment's bounds register.
3. **Managing Free Physical Memory:** This is the most significant challenge introduced by segmentation.
    - **External Fragmentation:** As segments of various sizes are allocated and freed, physical memory becomes littered with small, unusable holes of free space. This is called **external fragmentation**.

- **The Problem:** A new segment may not fit into any of the available holes, even if the total amount of free memory is sufficient.
- **Solutions (with drawbacks):**
  - **Compaction:** The OS can stop all processes and move existing segments together to create one large free block. This is extremely slow and computationally expensive.
  - **Free-List Algorithms:** Use smart algorithms (like best-fit, worst-fit, first-fit) to manage the free space and try to minimize fragmentation. However, no algorithm can eliminate it entirely.



5. Summary

- **Advantages of Segmentation:**
  - It efficiently supports sparse address spaces by not allocating memory for the unused gap between the heap and stack.
  - It enables code sharing, saving physical memory.
  - Address translation is fast with hardware support.
- **Disadvantages of Segmentation:**
  - It suffers from the **external fragmentation** problem, which makes memory management complex and potentially inefficient.
  - It is not flexible enough. If a single segment (like a sparse heap) becomes very large, the entire segment must still be in physical memory, even if only small parts of it are being used.

These limitations show that while segmentation is a major improvement over simple base-and-bounds, an even more flexible solution is needed. This leads to the next major memory management technique: **paging**.