Chapter 20: Paging: Smaller Tables

This chapter addresses a major drawback of simple (linear) paging: page tables can become enormous, consuming a significant amount of memory. We will explore several techniques designed to reduce the memory footprint of page tables.

> **The Crux of the Problem: How to Make Page Tables Smaller?**
>
> Simple array-based page tables (usually called linear page tables) are too big, taking up far too much memory on typical systems. How can we make page tables smaller? What are the key ideas? What inefficiencies arise as a result of these new data structures?

1. Simple Solution: Bigger Pages

One straightforward way to reduce page table size is to increase the size of each page.

- **Mechanism:** If you increase the page size by a factor of N, you decrease the number of pages in the address space by the same factor, thus reducing the number of page table entries (PTEs) needed.
- **Example:** A 32-bit address space with 4KB pages needs a 4MB page table. Increasing the page size to 16KB reduces the page table size to 1MB.
- **The Problem: Internal Fragmentation:** The primary drawback is that larger pages lead to more wasted memory *within* each page. If a process only needs a few bytes on a 16KB page, the remaining space is wasted. Because of this, most systems use relatively small page sizes (e.g., 4KB) for general-purpose use.

---

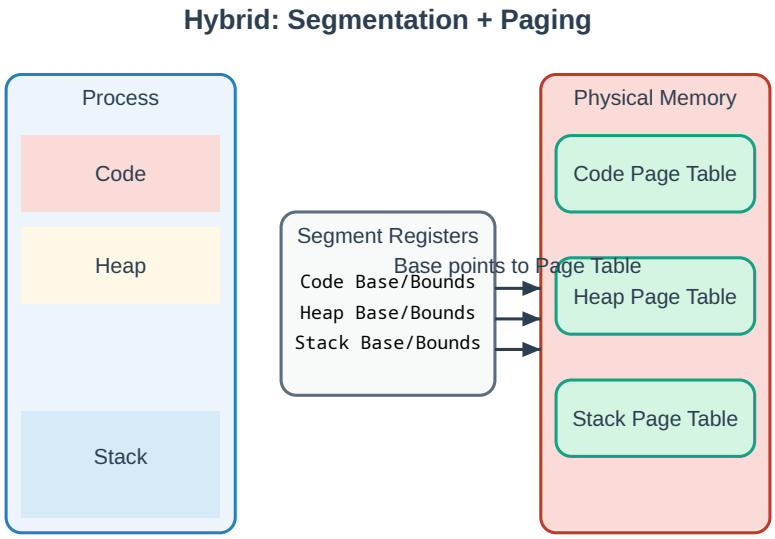
2. Hybrid Approach: Paging and Segments

This approach combines segmentation and paging to get the benefits of both.

- **The Problem with Linear Page Tables:** A linear page table must allocate space for the *entire* virtual address space, even the large, unused gap between the heap and stack. This space is filled with invalid PTEs, wasting memory.
- **Hybrid Idea:** Instead of one large page table for the whole address space, use one smaller page table for each logical *segment* (code, heap, stack).
- **Mechanism:**
  1. The hardware has a base/bounds register pair for each segment (e.g., code base/bounds, heap base/bounds).

2. The **base register** for a segment does *not* point to the segment's data in physical memory. Instead, it points to the physical address of that segment's **page table**.
3. The **bounds register** checks if a virtual page number is valid for that segment's page table.

- **Benefit:** This approach avoids allocating page table entries for the large, unused region between the heap and stack, saving significant memory.
- **Drawbacks:**
  - It still relies on segmentation, which can be inflexible if a program's memory usage doesn't fit the code/heap/stack model well (e.g., a large, sparse heap).
  - It re-introduces **external fragmentation**, as the page tables themselves are now variable-sized chunks that must be allocated in physical memory.



3. Multi-Level Page Tables

This is the most common and effective solution used in modern systems (e.g., x86). It avoids the problems of segmentation by treating the page table itself as a paged data structure.

- **Core Idea:** Chop the linear page table into page-sized units. If a page of PTEs contains only invalid entries, don't allocate that page at all.
- **Mechanism:**
  1. A new structure called the **Page Directory** is introduced. The page directory contains one **Page Directory Entry (PDE)** for each page

of the page table.

2. A PDE contains a valid bit and a Physical Frame Number (PFN).
   - If a PDE is **valid**, its PFN points to the physical frame containing the corresponding page of PTEs.
   - If a PDE is **invalid**, it means the entire page of PTEs it would point to is unused, and thus no memory is allocated for it.

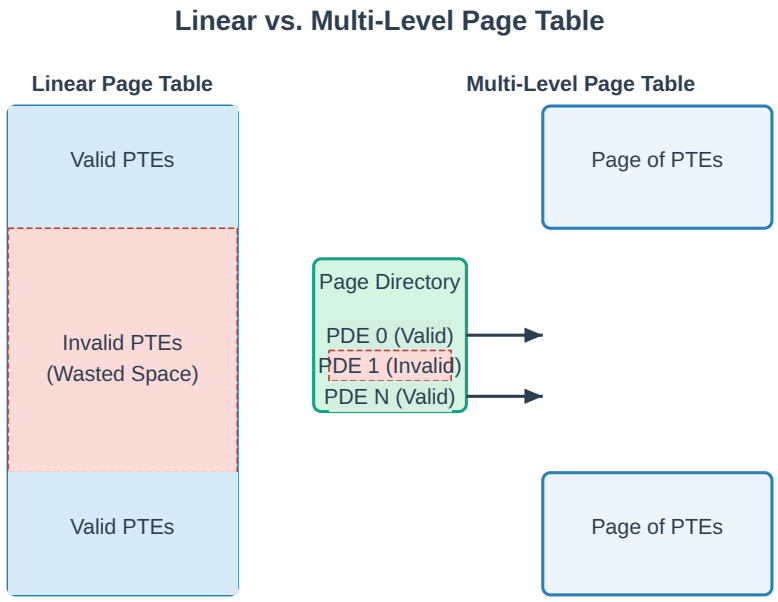


Figure 1: Linear vs. Multi-Level Page Table

3.1. Address Translation with a Two-Level Page Table

The virtual address is now split into three parts:

1. **Page Directory Index (PD Index):** The top bits of the VPN, used to index into the page directory.
2. **Page Table Index (PT Index):** The next set of bits, used to index into the page of PTEs.
3. **Offset:** The lowest bits, which remain unchanged.

- **Translation Process (on a TLB miss):**
  1. Use the **PD Index** to find the correct PDE in the page directory.
  2. Check if the PDE is valid. If not, it's a fault.
  3. If valid, use the PFN from the PDE to find the physical address of the correct page of the page table.

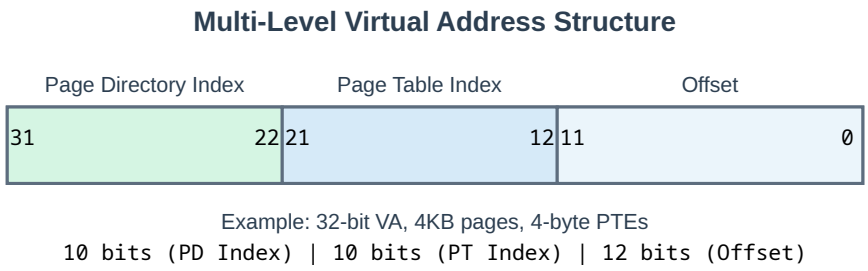


Figure 2: Multi-Level Virtual Address Structure

4. Use the **PT Index** to find the correct PTE within that page table page.
5. Check if the PTE is valid. If not, it's a fault.
6. If valid, use the PFN from the PTE to construct the final physical address by combining it with the offset.

3.2. Advantages and Disadvantages

- **Advantages:**
  - **Space Efficient:** Only allocates page table space proportional to the amount of address space in use. It elegantly supports sparse address spaces.
  - **Easy Allocation:** Each part of the page table (the page directory and its pages) fits neatly into a page frame, simplifying memory management for the OS.
- **Disadvantages:**
  - **Performance Cost:** On a TLB miss, a two-level lookup requires **two** memory accesses (one for the PDE, one for the PTE) instead of one. This is a classic **time-space trade-off**.
  - **Complexity:** The lookup process is more complex for the hardware or OS to implement.

3.3. More Than Two Levels

- If the page directory itself becomes too large to fit on a single page, the same trick can be applied again: page the page directory and create another level of indirection.
- Modern 64-bit systems often use three, four, or even five levels of page tables to manage the enormous virtual address space.

---

4. Other Approaches

- **Inverted Page Tables:** Instead of one page table per process, the system has a single page table with one entry for each *physical frame* of memory. The entry specifies which process and which virtual page are using that frame. This saves immense space but makes lookups difficult, often requiring a hash table to be efficient.
- **Swapping Page Tables to Disk:** In some systems, the page tables themselves can be swapped to disk, just like user pages. This is an advanced technique for systems under extreme memory pressure.

---

5. Summary

Simple linear page tables are too large for modern systems.

- **Bigger pages** can help but cause internal fragmentation.
- **Hybrid segmentation/paging** saves space but is inflexible and reintroduces external fragmentation.
- **Multi-level page tables** are the dominant solution. They are space-efficient and elegantly support sparse address spaces, at the cost of increased complexity and slower TLB miss handling. This time-space trade-off is acceptable because TLB hits are the common case.