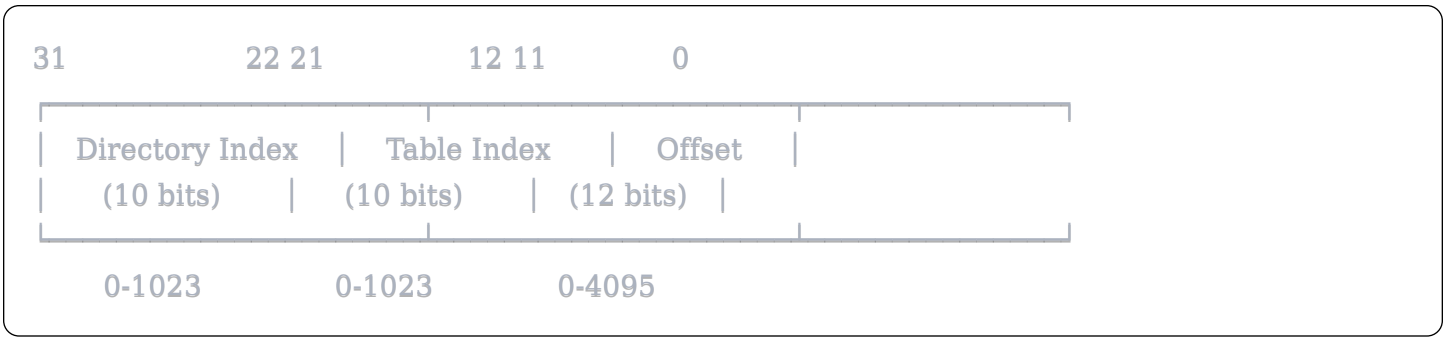
x86 Paging - Complete Guide

Virtual Address Structure (32 bits)



Address Translation

```
Virtual 0xC0100500
Binary: 11000000 00010000 00000101 00000000

Bits 31-22: 11 0000 0000 = 768  (Directory Index)
Bits 21-12: 00 0100 0000 = 256  (Table Index)
Bits 11-0:  0101 0000 0000 = 0x500 (Offset)

Translation:
1. Look up kernel_page_directory[768] → Get address of page table
2. Look up page_table[256] → Get physical page address
3. Add offset: physical_page + 0x500 = final physical address
```

Page Directory

Purpose: Stores addresses of page tables

Size: 4KB (1024 entries × 4 bytes)

Each entry covers: 4MB of virtual address space

```
Index  Virtual Address Range    Points To
──────────────────────────────────────────
0      0x00000000 - 0x003FFFFF  Address of first_table
1      0x00400000 - 0x007FFFFF  Address of second_table (or NULL)
2      0x00800000 - 0x00BFFFFF  NULL
...
768    0xC0000000 - 0xC03FFFFF  Address of first_table (same as [0]!)
769    0xC0400000 - 0xC07FFFFF  Address of another table
...
1023   0xFFC00000 - 0xFFFFFFFF  NULL
```

Why index determines virtual range: CPU extracts bits 31-22 from any virtual address to get the directory index.

Page Table

Purpose: Stores physical addresses of actual memory pages
Size: 4KB (1024 entries × 4 bytes)
Each entry covers: 4KB of memory

```
Index  Physical Address
_____________________________
0     0x00000003  (phys: 0x00000000 + flags)
1     0x00001003  (phys: 0x00001000 + flags)
2     0x00002003  (phys: 0x00002000 + flags)
...
256   0x00100003  (phys: 0x00100000 + flags)  ← Kernel here
...
1023  0x003FF003  (phys: 0x003FF000 + flags)
```

Entry format: Bottom 12 bits = flags, Top 20 bits = physical page address

Example Setup: Identity Mapping

```c
// Allocate page table (gets 4KB of physical memory)
uint32_t *first_table = (uint32_t*)pmm_alloc(1);

// Fill table: Map virtual page N → physical page N
for (int i = 0; i < 1024; i++) {
    uint32_t physical_addr = i * 0x1000;  // Physical address
    first_table[i] = physical_addr | PAGE_PRESENT | PAGE_WRITE;
}

// Link table to page directory
kernel_page_directory[0] = ((uint32_t)first_table) | PAGE_PRESENT | PAGE_WRITE;
```

What this does:

- Creates virtual addresses 0x00000000 - 0x003FFFFF (because we used directory[0])
- Maps them to physical addresses 0x00000000 - 0x003FFFFF
- Virtual = Physical (identity mapping)

Higher-Half Kernel Mapping

```c
// Map SAME table at index 768
kernel_page_directory[768] = ((uint32_t)first_table) | PAGE_PRESENT | PAGE_WRITE;
```

Result: Now the SAME physical memory (0-4MB) is accessible via TWO virtual ranges:

- Virtual 0x00000000 - 0x003FFFFF (via directory[0])
- Virtual 0xC0000000 - 0xC03FFFFF (via directory[768])

Both point to physical 0x00000000 - 0x003FFFFF.

Complete Example

Page Directory (at physical 0x9000)

Index	Value	What it does
0	0x00001003 (first_table)	Virtual 0x00000000-0x003FFFFF → Use first_table for lookup
768	0x00001003 (first_table)	Virtual 0xC0000000-0xC03FFFFF → Use first_table for lookup

Page Table: first_table (at physical 0x1000)

Index	Value	Physical Address
0	0x00000003	0x00000000
1	0x00001003	0x00001000
256	0x00100003	0x00100000 (kernel!)
1023	0x003FF003	0x003FF000

Translation Examples

Virtual 0xC0100500 → Physical ?

```
Step 1: Extract bits
  Directory Index = 768 (bits 31-22)
  Table Index = 256 (bits 21-12)
  Offset = 0x500 (bits 11-0)

Step 2: kernel_page_directory[768] = 0x00001003
  → first_table is at physical 0x00001000

Step 3: first_table[256] = 0x00100003
  → Physical page at 0x00100000

Step 4: Physical address = 0x00100000 + 0x500 = 0x00100500
```

Virtual 0x00100500 → Physical ?

```
Step 1: Extract bits
  Directory Index = 0
  Table Index = 256
  Offset = 0x500

Step 2: kernel_page_directory[0] = 0x00001003
  → first_table is at physical 0x00001000 (SAME TABLE!)

Step 3: first_table[256] = 0x00100003
  → Physical page at 0x00100000 (SAME RESULT!)

Step 4: Physical address = 0x00100000 + 0x500 = 0x00100500
```

Both virtual addresses reach the same physical memory!

Key Points

1. **Page Directory entries**: Store addresses of page tables
2. **Page Table entries**: Store physical addresses of actual memory
3. **Virtual address bits 31-22**: Automatically select page directory index
4. **Virtual address bits 21-12**: Automatically select page table index
5. **You create virtual addresses**: By choosing which directory/table indices to fill
6. **CPU does translation**: You never manually convert - hardware does it automatically

Memory Tracking

You need TWO separate tracking systems:

Physical Memory Manager (PMM)

- Tracks which physical frames are free/used
- Returns physical addresses when you call `pmm_alloc()`

Virtual Memory Manager (VMM)

- Tracks which virtual addresses are allocated
- Updates page tables when allocating virtual memory
- Uses bitmap or next-free-pointer to track virtual space

c

```c
// Example allocation flow:
void* vmm_alloc_pages(size_t count) {
    // 1. Pick free virtual addresses (from your tracking)
    uint32_t virt_addr = next_free_virtual_addr;

    // 2. Get physical frames for each page
    for (size_t i = 0; i < count; i++) {
        uint32_t phys_frame = pmm_alloc(1);  // Physical address
        uint32_t virt_page = virt_addr + (i * 0x1000);

        // 3. Map virtual → physical in page tables
        map_page(virt_page, phys_frame, PAGE_PRESENT | PAGE_WRITE);
    }

    // 4. Update virtual address tracker
    next_free_virtual_addr += count * 0x1000;

    return (void*)virt_addr;
}
```

Enabling Paging

c

```c
// 1. Set CR3 to page directory physical address
asm volatile("mov %0, %%cr3" : : "r"(kernel_page_directory));

// 2. Enable paging bit in CR0
uint32_t cr0;
asm volatile("mov %%cr0, %0" : "=r"(cr0));
cr0 |= 0x80000000;  // Set PG bit
asm volatile("mov %0, %%cr0" : : "r"(cr0));
```

Now all memory accesses go through page tables!