Case studies: x86, xv6, Linux

Diagrams from:
- AMD64 Architecture Programmer’s Manual, Volume 2: System Programming
- xv6 Commentary
x86 VM support

- x86 (aka IA-32) supports segmentation & paging in 32-bit protected mode
- x86-64 (aka IA-32e) introduces 64-bit (nominal) mode
  - Segmentation is mostly deprecated in favor of paging
- Support for coexisting normal and “huge” pages
32-bit segmentation + paging

PROTECTED-MODE MEMORY MANAGEMENT

The segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment). The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a linear address in the processor's linear address space.

If paging is not used, the linear address space of the processor is mapped directly into the physical address space of processor. The physical address space is defined as the range of addresses that the processor can generate on its address bus.

Because multitasking computing systems commonly define a linear address space much larger than it is economically feasible to contain all at once in physical memory, some method of "virtualizing" the linear address space is needed. This virtualization of the linear address space is handled through the processor's paging mechanism.

Paging supports a "virtual memory" environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk.
32-bit segmentation

Logical Address 15 0
Seg. Selector 31(63) Offset (Effective Address) 0

Descriptor Table
Segment Descriptor
Base Address +

Segment descriptor format

Base 31:24     D / L     A / V     Seg. Limit 19:16     P     D / P     S     Type     Base 23:16
31  24 23 22 21 20 19  16 15 14 13 12 11  8 7 0

Segment Limit 15:00

Segmentation registers

Visible Part

Hidden Part

<table>
<thead>
<tr>
<th>Segment Selector</th>
<th>Base Address, Limit, Access Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
</tr>
</tbody>
</table>

L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment_limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type
The flags and fields in a segment descriptor are as follows:

- **Segment limit field**: Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:
  - If the granularity flag is clear, the segment size can range from 1 byte to 1 MByte, in byte increments.
  - If the granularity flag is set, the segment size can range from 4 KBytes to 4 GBytes, in 4-KByte increments.

- **Base address fields**: Defines the location of byte 0 of the segment within the 4-GByte linear address space. The processor puts together the three base address fields to form a single 32-bit value. Segment base addresses should be aligned to 16-byte boundaries. Although 16-byte alignment is not required, this alignment allows programs to maximize performance by aligning code and data on 16-byte boundaries.

- **Type field**: Indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an application (code or data) descriptor or a system descriptor. The encoding of the type field is different for code, data, and system descriptors (see Figure 5-1). See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types", for a description of how this field is used to specify code and data-segment types.

---

### Segment descriptor format

```c
struct segdesc {
    uint lim_15_0 : 16; // Low bits of segment limit
    uint base_15_0 : 16; // Low bits of segment base address
    uint base_23_16 : 8; // Middle bits of segment base address
    uint type : 4; // Segment type (see STS_ constants)
    uint s : 1; // 0 = system, 1 = application
    uint dpl : 2; // Descriptor Privilege Level
    uint p : 1; // Present
    uint lim_19_16 : 4; // High bits of segment limit
    uint avl : 1; // Unused (available for software use)
    uint rsv1 : 1; // Reserved
    uint db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
    uint g : 1; // Granularity: limit scaled by 4K when set
    uint base_31_24 : 8; // High bits of segment base address
};
```

#define SEG(type, base, lim, dpl) (struct segdesc)    
{(lim) >> 12) & 0xffff, (uint)(base) & 0xffff,    
(C(uint)(base) >> 16) & 0xff, type, 1, dpl, 1,    
(uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24}

---

L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type
32-bit xv6 segment initialization

```c
struct segdesc {
    uint lim_15_0 : 16; // Low bits of segment limit
    uint base_15_0 : 16; // Low bits of segment base address
    uint base_23_16 : 8; // Middle bits of segment base address
    uint type : 4; // Segment type (see STS constants)
    uint s : 1; // 0 = system, 1 = application
    uint dpl : 2; // Descriptor Privilege Level
    uint p : 1; // Present
    uint lim_19_16 : 4; // High bits of segment limit
    uint avl : 1; // Unused (available for software use)
    uint rsv1 : 1; // Reserved
    uint db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
    uint g : 1; // Granularity: limit scaled by 4K when set
    uint base_31_24 : 8; // High bits of segment base address
};

#define SEG(type, base, lim, dpl) (struct segdesc)    
    { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, 
    (uint)(base) >> 16) & 0xff, type, 1, dpl, 1, 
    (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 } 

void seginit(void)
{
    struct cpu *c;
    c = &cpus[cpuid()];
    c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
    c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
    c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
    c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
    lgdt(c->gdt, sizeof(c->gdt));
}
```
Flat vs. Multi-segment models

Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels. Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.
32-bit 4KB vs. 4MB pages
32-bit 4KB xv6 page table walk/alloc

```
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    pde_t *pde;
    pte_t *pgtab;

    pde = &pgdir[PDX(va)];
    if(*pde & PTE_P){
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
    } else {
        if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
            return 0;
        memset(pgtab, 0, PGSIZE);
        *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
    }
    return &pgtab[PTX(va)];
}
```
Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are “not present”; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.

### CR3 and paging structure entries

<table>
<thead>
<tr>
<th>Address of page directory</th>
<th>Ignored</th>
<th>P</th>
<th>C</th>
<th>D</th>
<th>W</th>
<th>Ignored</th>
<th>CR3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 31:22 of address of 2MB page frame</td>
<td>Reserved (must be 0)</td>
<td>Bits 39:32 of address</td>
<td>P A T</td>
<td>Ignored</td>
<td>G</td>
<td>O</td>
<td>D</td>
</tr>
<tr>
<td>Address of page table</td>
<td>Ignored</td>
<td>O</td>
<td>I</td>
<td>g</td>
<td>A</td>
<td>P C D</td>
<td>W</td>
</tr>
<tr>
<td>Ignored</td>
<td>0</td>
<td>PDE: not present</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address of 4KB page frame</td>
<td>Ignored</td>
<td>G</td>
<td>P A T</td>
<td>D</td>
<td>A</td>
<td>P C D</td>
<td>W</td>
</tr>
<tr>
<td>Ignored</td>
<td>0</td>
<td>PTE: not present</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4KB PTE breakdown

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to map a 4-KByte page</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>6 (D)</td>
<td>Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>7 (PAT)</td>
<td>If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)</td>
</tr>
<tr>
<td>8 (G)</td>
<td>Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise</td>
</tr>
<tr>
<td>11:9</td>
<td>Ignored</td>
</tr>
<tr>
<td>31:12</td>
<td>Physical address of the 4-KByte page referenced by this entry</td>
</tr>
</tbody>
</table>
xv6 paging structure initialization

```c
#define EXTMEM 0x100000 // Start of extended memory
#define PHYSTOP 0xE000000 // Top physical memory
#define DEVSPACE 0xFE000000 // Other devices are at high addresses
#define KERNBASE 0x80000000 // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked
#define PGSIZE 4096 // bytes mapped by a page
#define PGROUNDUP(sz) (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))

// This table defines the kernel's mappings, present in every process
static struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
} kmap[] = {
    { (void*)KERNBASE, 0, EXTMEM, PTE_W}, // I/O space
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
    { (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern data+memory
    { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
};
```

```c
static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm) {
    char *a, *last;
    pte_t *pte;

    a = (char*)PGROUNDDOWN((uint)va);
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
    for(;;){
        if((pte = walkpgdir(pgdir, a, 1)) == 0)
            return -1;
        if(*pte & PTE_P)
            panic("remap");
        *pte = pa | perm | PTE_P;
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}
```
// Set up kernel part of a page table.
pde_t* setupkvm(void) {
    pde_t *pgdir;
    struct kmap *k;

    pgdir = (pde_t*)kalloc();
    memset(pgdir, 0, PGSIZE);

    if (P2V(PHYSTOP) > (void*)DEVSPACE)
        panic("PHYSTOP too high");

    for (k = kmap; k < &kmap[NELEM(kmap)]; k++)
        mappages(pgdir, k->virt, k->phys_end - k->phys_start, (uint)k->phys_start, k->perm);

    return pgdir;
}

#define EXTMEM 0x100000  // Start of extended memory
#define PHYSTOP 0xE000000  // Top physical memory
#define DEVSPACE 0xFE000000  // Other devices are at high addresses
#define KERNBASE 0x80000000  // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
#define PGSIZE 4096  // bytes mapped by a page
#define PGROUNDUP(sz) (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))

// This table defines the kernel's mappings, present in every process
static struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
} kmap[] = {
    (void*)KERNBASE, 0, EXTMEM, PTE_W, // I/O space
    (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0, // kern text+rodata
    (void*)data, V2P(data), PHYSTOP, PTE_W, // kern data+memory
    (void*)DEVSPACE, DEVSPACE, 0, PTE_W, // more devices
};
# Allocate page tables and physical memory to grow process

```c
int allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    char *mem;
    uint a;

    if (newsz >= KERNBASE) return 0;
    if (newsz < oldsz) return oldsz;

    a = PGROUNDUP(oldsz);
    for (; a < newsz; a += PGSIZE){
        mem = kalloc();
        memset(mem, 0, PGSIZE);
        mappages(pgdir, ((char *)a), PGSIZE, V2P(mem), PTE_W|PTE_U);
    }
    return newsz;
}
```

### Macros

```c
#define EXTMEM 0x100000 // Start of extended memory
#define PHYSTOP 0x80000000 // Top physical memory
#define DEVSPACE 0xFE000000 // Other devices are at high addresses
#define KERNBASE 0x80000000 // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked
#define KERNBASE+EXTMEM // First kernel virtual address
#define PGSIZE 4096 // bytes mapped by a page
#define PGROUNDUP(sz) (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))
```

### Static kmap table

```c
static struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
} kmap[] = {
    (void*)KERNBASE, 0, EXTMEM, PTE_W, // I/O space
    (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0, // kern text+rodata
    (void*)data, V2P(data), PHYSTOP, PTE_W, // kern data+memory
    (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
};
```
xv6 paging structure initialization

```c
#define EXTMEM 0x1000000 // Start of extended memory
#define PHYSTOP 0xE0000000 // Top physical memory
#define DEVSPACE 0xFE0000000 // Other devices are at high addresses
#define KERNBASE 0x80000000 // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked

#define PGSIZE 4096 // bytes mapped by a page
#define PGROUNDUP(sz) (((sz)+PGSIZE-1) & ~PGSIZE-1)
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))
```

```
// This table defines the kernel's mappings, present in every process.
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
  { (void*)KERNBASE, 0, EXTMEM, PTE_W}, // I/O space
  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
  { (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern data+memory
  { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
};
```
### Beyond 32-bit address spaces

The three paging modes differ with regard to the following details:

- **Linear-address width.** The size of the linear addresses that can be translated.
- **Physical-address width.** The size of the physical addresses produced by paging.
- **Page size.** The granularity at which linear addresses are translated. Linear addresses on the same page are translated to corresponding physical addresses on the same page.
- **Support for execute-disable access rights.** In some paging modes, software can be prevented from fetching instructions from pages that are otherwise readable.
- **Support for PCIDs.** With 4-level paging, software can enable a facility by which a logical processor caches information for multiple linear-address spaces. The processor may retain cached information when software switches between different linear-address spaces.
- **Support for protection keys.** With 4-level paging, software can enable a facility by which each linear address is associated with a protection key. Software can use a new control register to determine, for each protection keys, how software can access linear addresses associated with that protection key.

Table 4-1 illustrates the principal differences between the three paging modes.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>32</td>
<td>32</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>32-bit</td>
<td>1</td>
<td>0</td>
<td>0(^2)</td>
<td>32</td>
<td>Up to 40(^3)</td>
<td>4 KB 4 MB(^4)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PAE</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>32</td>
<td>Up to 52</td>
<td>4 KB 2 MB</td>
<td>Yes(^5)</td>
<td>No</td>
</tr>
<tr>
<td>4-level</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>48</td>
<td>Up to 52</td>
<td>4 KB 2 MB 1 GB(^6)</td>
<td>Yes(^5)</td>
<td>Yes(^7)</td>
</tr>
</tbody>
</table>

### Notes:

1. The physical-address width is always bounded by MAXPHYADDR; see Section 4.1.4.
2. The LMA flag in the IA32_EFER MSR (bit 10) is a status bit that indicates whether the logical processor is in IA-32e mode (and thus using 4-level paging). The processor always sets IA32_EFER.LMA to CR0.PG & IA32_EFER.LME. Software cannot directly modify IA32_EFER.LMA; an execution of WRMSR to the IA32_EFER MSR ignores bit 10 of its source operand.
3. 32-bit paging supports physical-address widths of more than 32 bits only for 4-MByte pages and only if the PSE-36 mechanism is supported; see Section 4.1.4 and Section 4.3.
4. 4-MByte pages are used with 32-bit paging only if CR4.PSE = 1; see Section 4.3.
5. Execute-disable access rights are applied only if IA32_EFER.NXE = 1; see Section 4.6.
6. Not all processors that support 4-level paging support 1-GByte pages; see Section 4.1.4.
7. PCIDs are used only if CR4.PCIDE = 1; see Section 4.10.1. Protection keys are used only if certain conditions hold; see Section 4.6.2.

### Table 4-1: Properties of Different Paging Modes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>32</td>
<td>32</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>32-bit</td>
<td>1</td>
<td>0</td>
<td>0(^2)</td>
<td>32</td>
<td>Up to 40(^3)</td>
<td>4 KB 4 MB(^4)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PAE</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>32</td>
<td>Up to 52</td>
<td>4 KB 2 MB</td>
<td>Yes(^5)</td>
<td>No</td>
</tr>
<tr>
<td>4-level</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>48</td>
<td>Up to 52</td>
<td>4 KB 2 MB 1 GB(^6)</td>
<td>Yes(^5)</td>
<td>Yes(^7)</td>
</tr>
</tbody>
</table>

### Notes:

1. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.
2. The processor ensures that IA32_EFER.LME must be 0 if CR0.PG = 1 and CR4.PAE = 0.
x86-64 (aka IA-32e) modes

- Long mode: 48-bit virtual addresses (256TB virtual address spaces)
  - 4-levels of paging structures
  - All but two segment registers are forced to a flat model, and no segment limit checking is performed
    - FS & GS segments can contain non-zero bases (useful for OS)
- Compatibility mode allows for 32-bit code to run unaltered
- Intel has started implementing 5-level paging to support 57-bit virtual addresses (as of Ice Lake)
Long mode paging

- 48-bit virtual addresses with 4 levels of paging
- Depending on paging structure entries, supports 4KB, 2MB, 1GB page sizes
Long mode 4KB paging

Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

- CR3
- PML4
- Directory Ptr
- Directory
- Table
- Offset

- PML4E
- PDPT

- 9

- PDE with PS=0
- PTE
- Page Table
- Physical Addr

- 4-KByte Page
Long mode 2MB paging

Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging
The following items describe the IA-32e paging process in more detail as well as how the page size is determined.

• A 4-KByte naturally aligned PML4 table is located at the physical address specified in bits 51:12 of CR3 (see Table 4-12). A PML4 table comprises 512 64-bit entries (PML4Es). A PML4E is selected using the physical address defined as follows:
  - Bits 51:12 are from CR3.
  - Bits 11:3 are bits 47:39 of the linear address.
  - Bits 2:0 are all 0.
  Because a PML4E is identified using bits 47:39 of the linear address, it controls access to a 512-GByte region of the linear-address space.

• A 4-KByte naturally aligned page-directory-pointer table is located at the physical address specified in bits 51:12 of the PML4E (see Table 4-14). A page-directory-pointer table comprises 512 64-bit entries (PDPTEs). A PDPTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PML4E.
Access control and metadata

- User/Supervisor and Read/Write flags in paging structure entries can be used to guard access
- If U/S flag = 0 (supervisor), can only access page if CPL = 0
- Accessed and Dirty flags are also useful for kernel swapping policies
Linux VM features

- Page cache and Sharing
- Swap cache
- Copy-on-write optimization
- Page allocation: buddy system
- Kernel internal memory management: slab allocator
Page cache and Sharing

- When executing a program (or loading shared libraries, etc.), the source file is not immediately loaded, but rather linked into the process’s virtual address space

- Page faults cause data to be loaded, a page at a time

- All file data loaded this way have entries in the page cache, which the kernel consults before going to disk

- If multiple processes access the same files, the kernel can share cached pages between them (potentially at different virtual addresses)

- **Dirty bit** needed to ensure that page isn’t modified
Page cache and Sharing

P₀ VM

Page cache

/bin/ls (1)
/bin/ls (2)
...
libc.so (1)
libc.so (2)
...
/usr/bin/vim (1)
/usr/bin/vim (2)
...

P₁ VM
Swap cache

- Dirty pages are swapped out (to save their contents) when low on memory
  - Unmodified pages can just be loaded from the page cache!
- **Swap cache** keeps track of pages that have been written to swap
  - If a page was previously swapped out and wasn’t modified after being swapped back in, can simply discard it
  - Helpful optimization for when system is heavily swapping (thrashing)
Copy-on-write (COW)

- “Clone” operation is quite common (e.g., used when fork-ing a process)

- But if carried out literally — duplicating entire memory image — is incredibly expensive (and likely unnecessary)

- At clone time, no data is actually copied; simply replicate paging structures and mark pages as read-only

- Page faults that occur on write accesses trigger copy operation
Page allocation

- Because pages are all the same size, theoretically we have no external fragmentation

- Can allocate first free page we find and map it into any virtual address space using paging structures

- But recall: large blocks of contiguous pages can be mapped as a single huge page (e.g., 4KB vs. 4MB)

- Can greatly improve TLB effectiveness!
  - Especially desirable given many levels of paging structures

- Also needed for I/O device direct memory access (more on this later)
Buddy system allocator

- Linux uses a “buddy system” allocator to search for blocks of free pages
- Idea: maintain separate lists of free page blocks, with sizes = powers of 2
  - E.g., list #0 = 1 page blocks,
    list #1 = 2 page blocks,
    list #2 = 4 page blocks,
    list #3 = 8 page blocks, etc.
- When allocating a block, keep splitting in half if possible
- When freeing a block, keep merging (doubling) if possible
Buddy system pros/cons

- Pros:
  - Fast allocation — search is easy
  - Able to find contiguous blocks
  - Good for huge pages
  - Can simplify page table updates

- Cons:
  - Small vs. Large blocks creates external fragmentation
  - \(2^n\) block sizes can result in significant internal fragmentation
  - Compromise: speed vs. efficiency
Kernel internal allocation

- Kernel frequently needs to free/allocate internal data structures
  - e.g., PCB entries, VM structures, file/inode structures
  - Fixed size, similarly initialized
  - Buddy allocator is not ideal — too much internal fragmentation!
  - Linux uses a **slab allocator** to allocate & free internal data structures
Slab allocator

- Built on top of the page buddy allocator
- Idea: allocate large blocks using buddy allocator, and carve them up into multiple data structure entries
  - Use the first one available, and leave partially initialized when freed
  - Effectively build dedicated caches for different data types
- Mitigates internal fragmentation due to buddy allocator