§ Case studies: x86, xv6, Linux

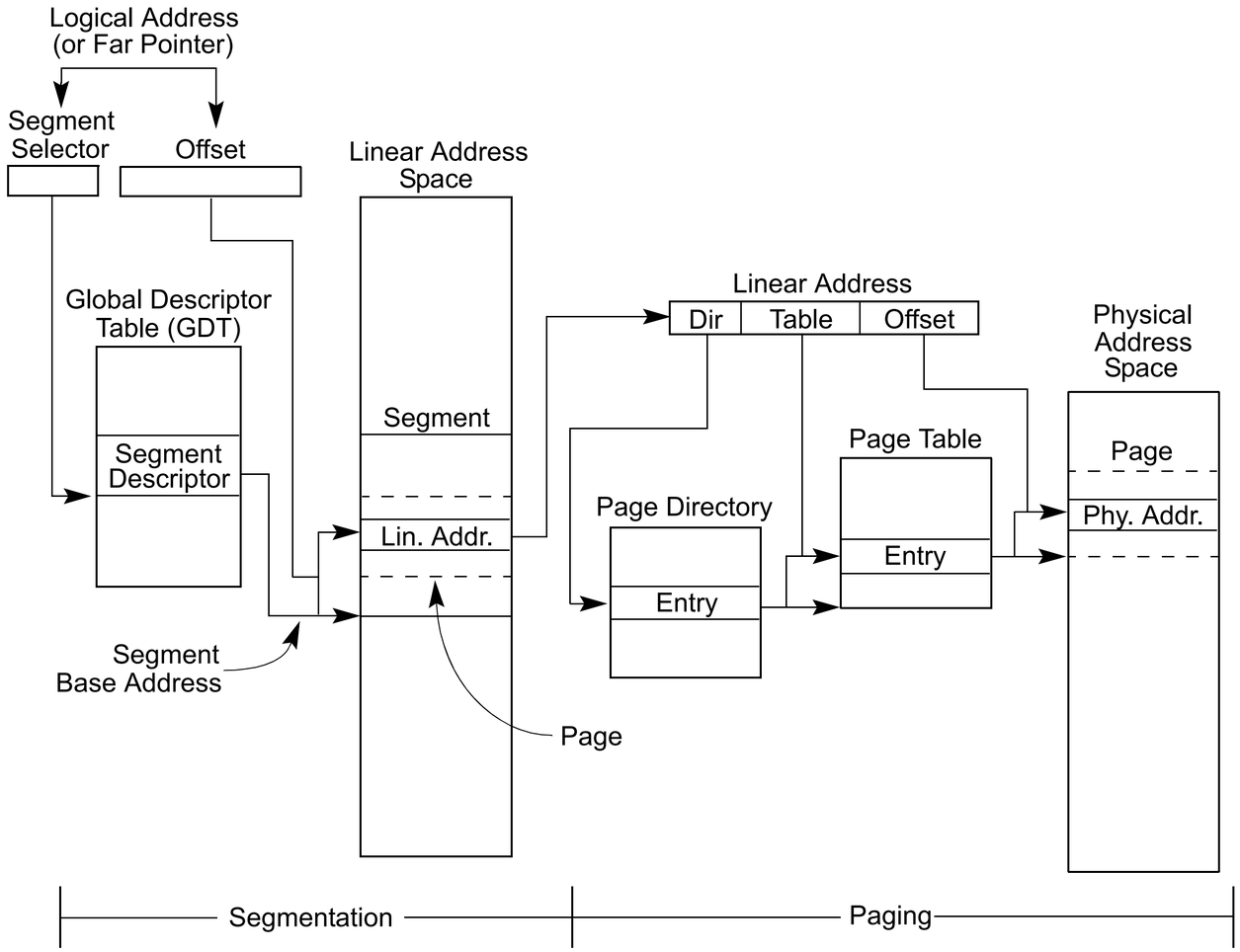
Diagrams from:

- Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide
- AMD64 Architecture Programmer's Manual, Volume 2: System Programming
- xv6 Commentary

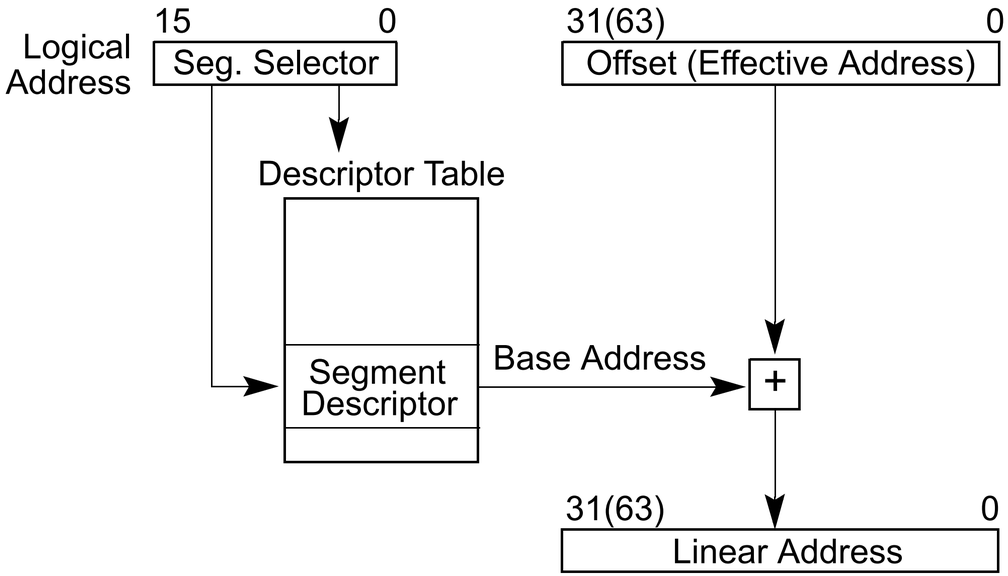
x86 VM support

- 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



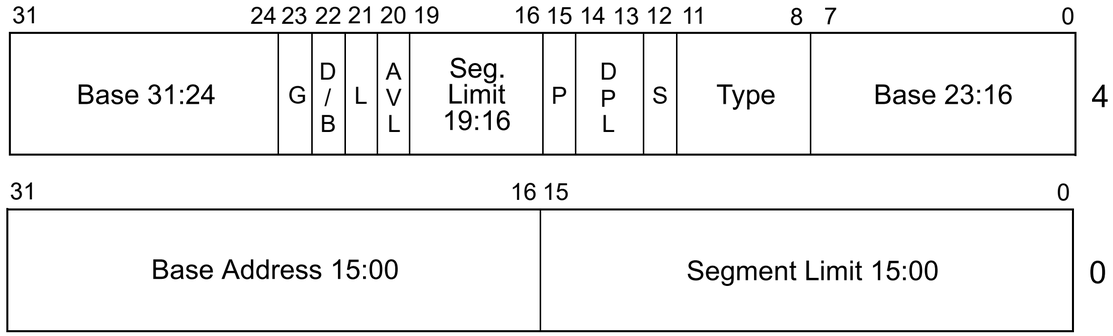
32-bit segmentation



Segmentation registers

Visible Part	Hidden Part	
Segment Selector	Base Address, Limit, Access Information	CS
		SS
		DS
		ES
		FS
		GS

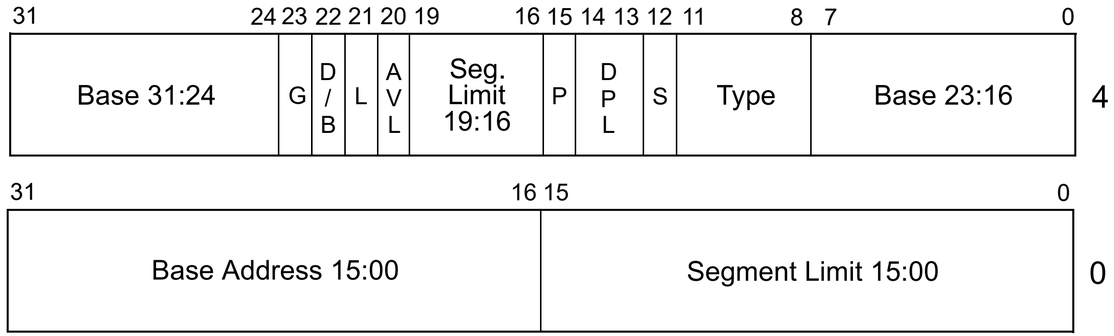
Segment descriptor format



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

Segment descriptor format



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

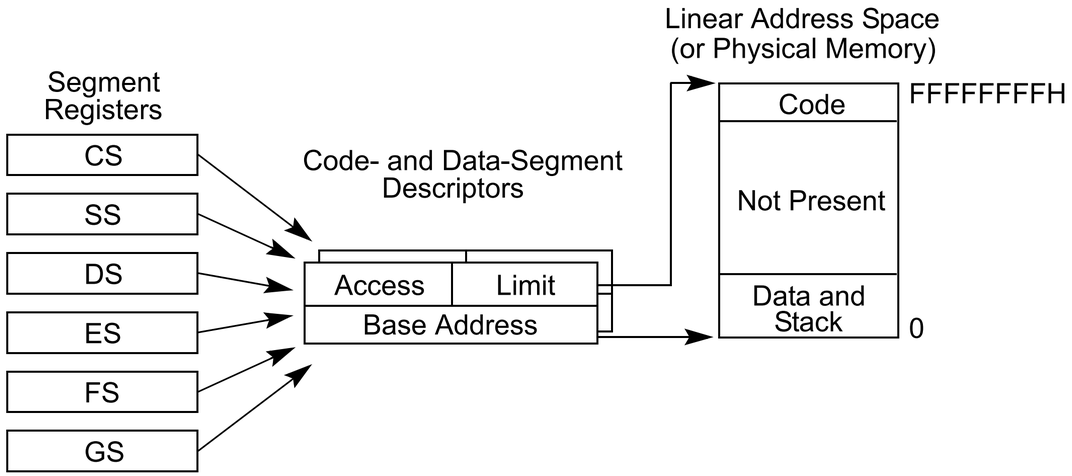
```
struct segdesc {
  uint lim_15_0 : 16;  // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4;       // Segment type (see STS_ constants)
  uint s : 1;          // 0 = system, 1 = application
  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 }
```

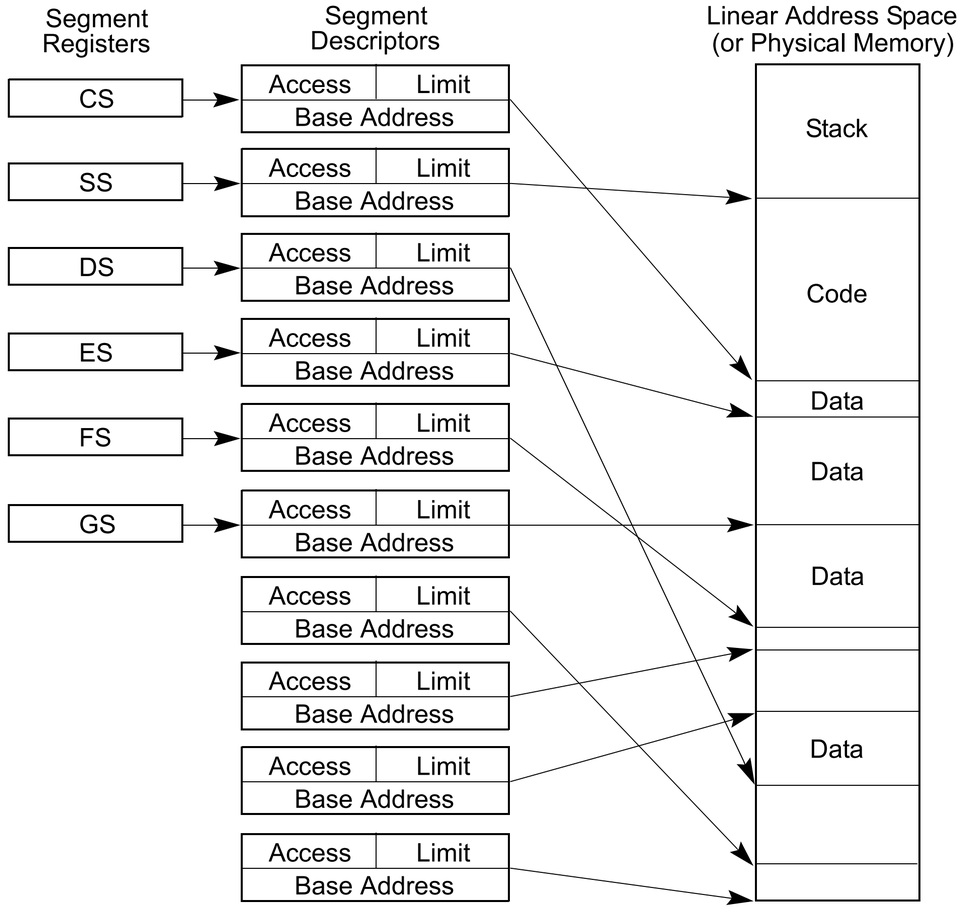
32-bit xv6 segment initialization

```
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 }
```

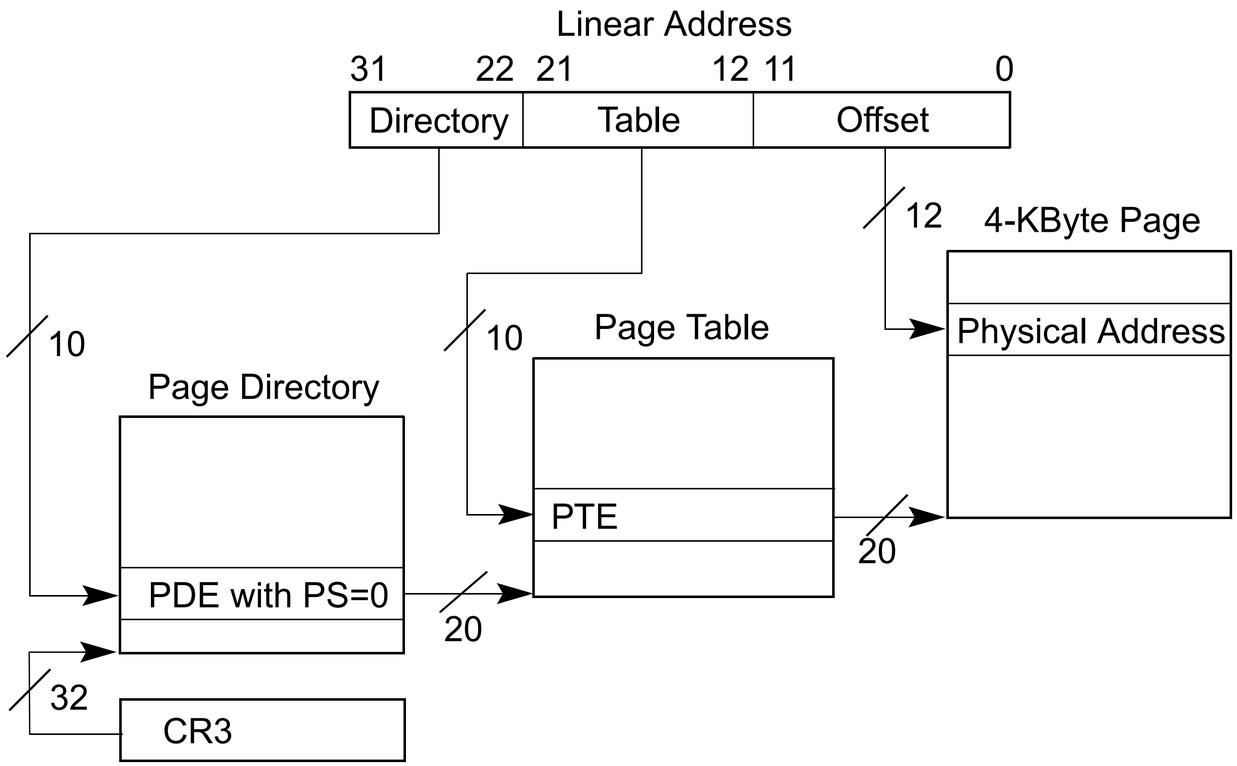
```
void
seginit(void)
{
  struct cpu *c;

  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));
}
```

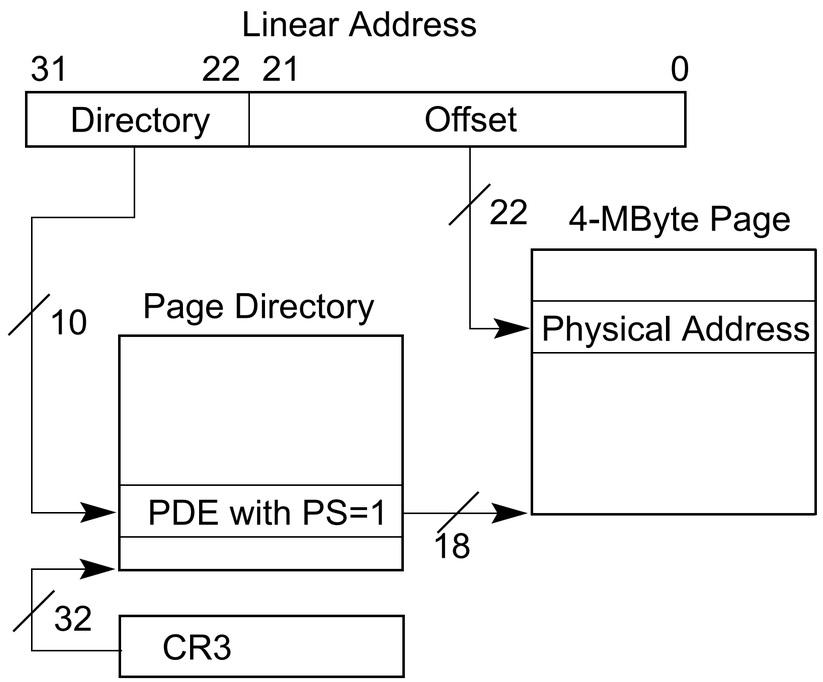
Flat vs. Multi-segment models



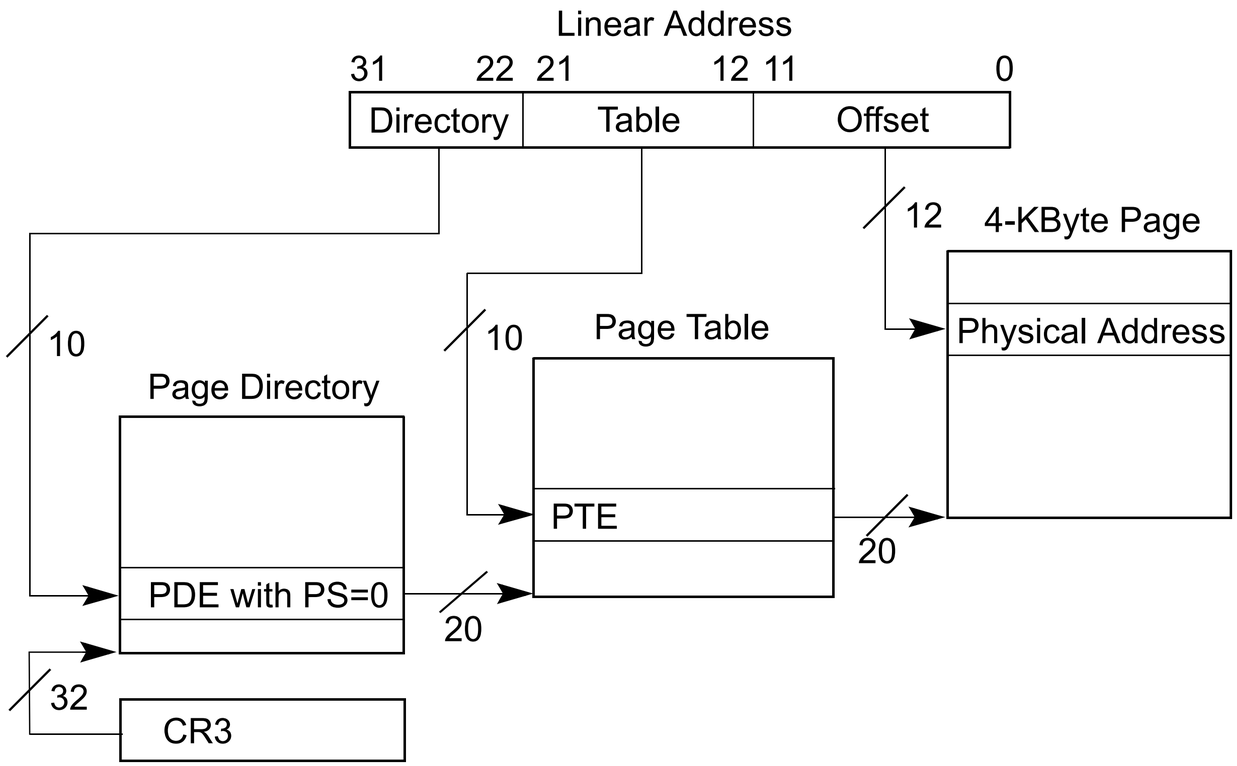


32-bit 4KB vs. 4MB pages





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)];
}
```

32-bit PDE/PTE

CR3 and paging structure entries

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored				G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

4KB PTE breakdown

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

xv6 paging structure initialization

```
#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
};
```

```
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;
}
```

xv6 paging structure initialization

```
#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
};
```

```
// 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;
}
```

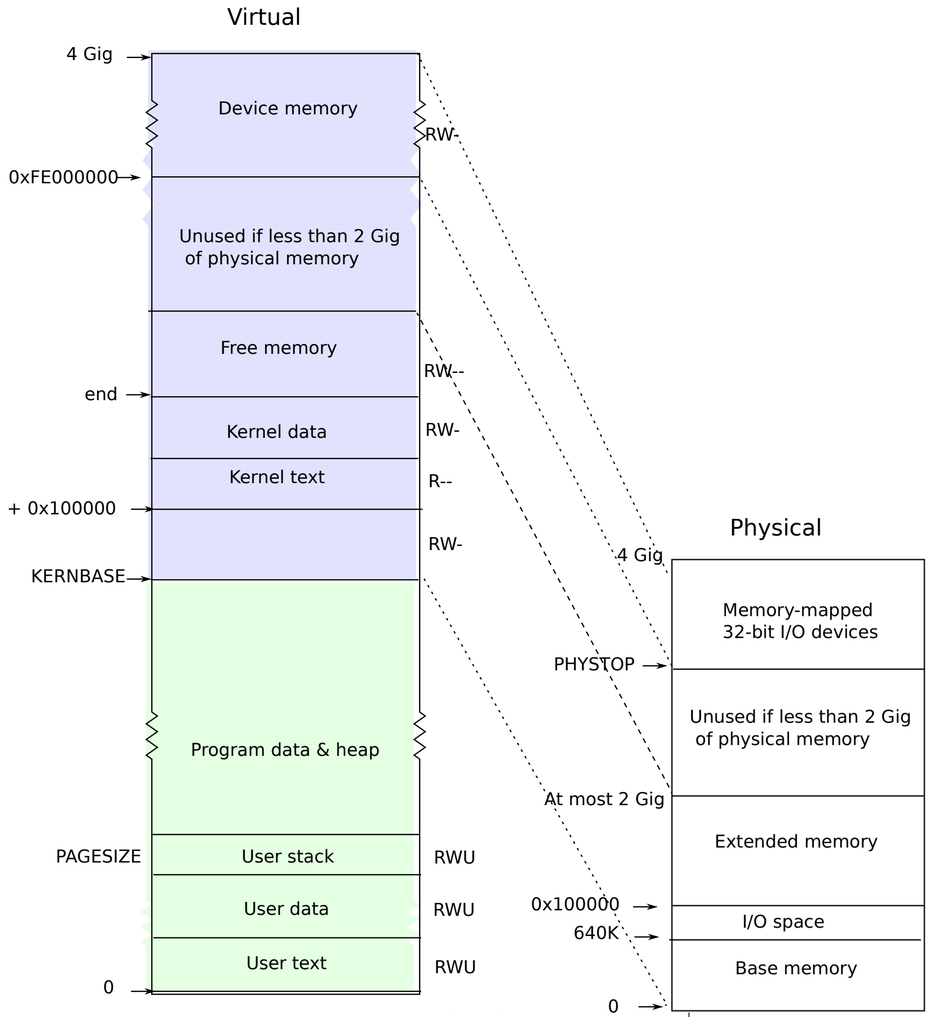
xv6 paging structure initialization

```
#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
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;
}
```

xv6 paging structure initialization

```
#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
};
```



Beyond 32-bit address spaces

Paging Mode	PG in CR0	PAE in CR4	LME in IA32_EFER	Lin.-Addr. Width	Phys.-Addr. Width[1]	Page Sizes	Supports Execute-Disable?	Supports PCIDs and protection keys?
None	0	N/A	N/A	32	32	N/A	No	No
32-bit	1	0	0[2]	32	Up to 40[3]	4 KB 4 MB[4]	No	No
PAE	1	1	0	32	Up to 52	4 KB 2 MB	Yes[5]	No
4-level	1	1	1	48	Up to 52	4 KB 2 MB 1 GB[6]	Yes[5]	Yes[7]

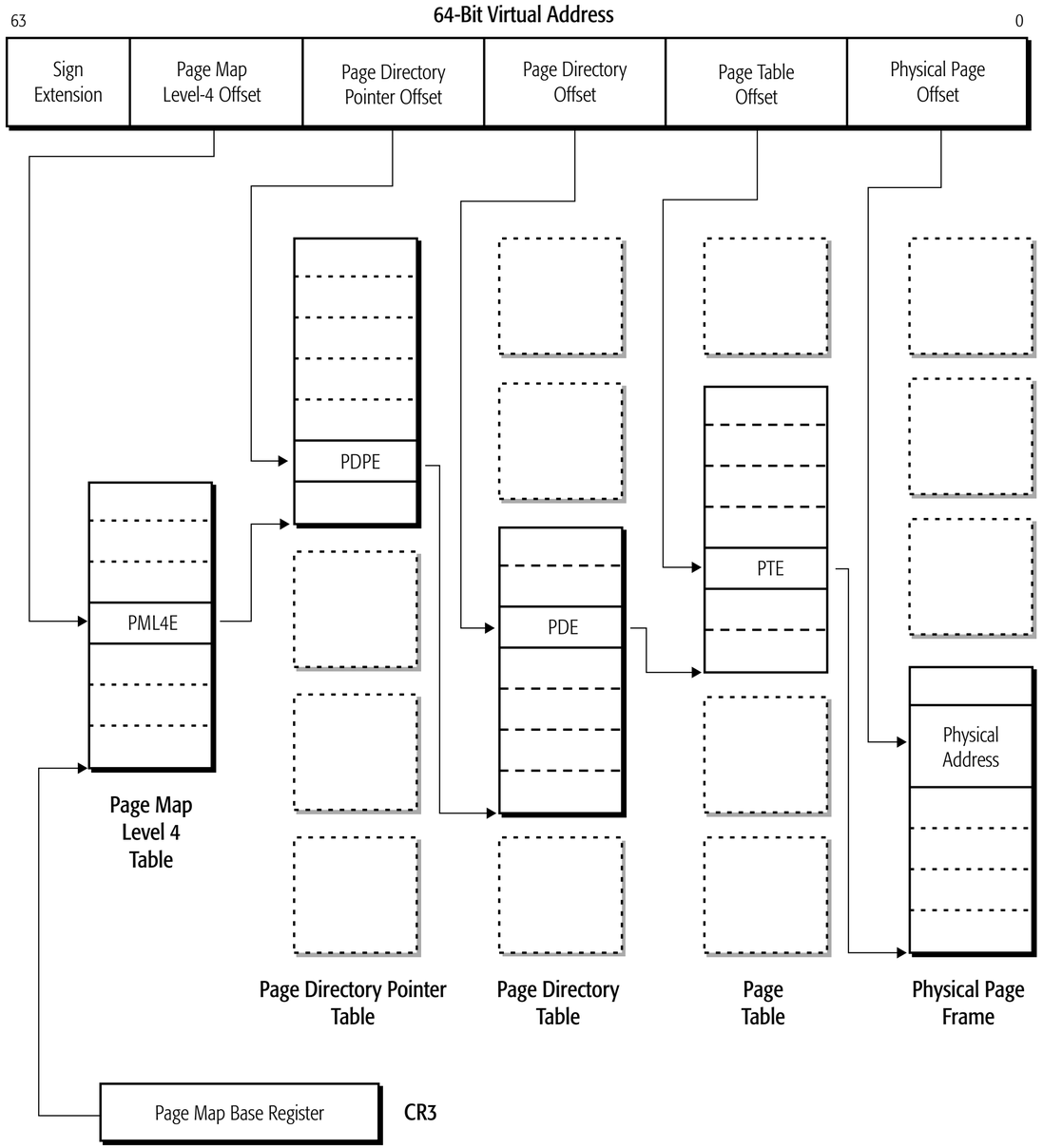
aka x86-64, per original AMD specification

x86-64 (aka IA-32e) modes

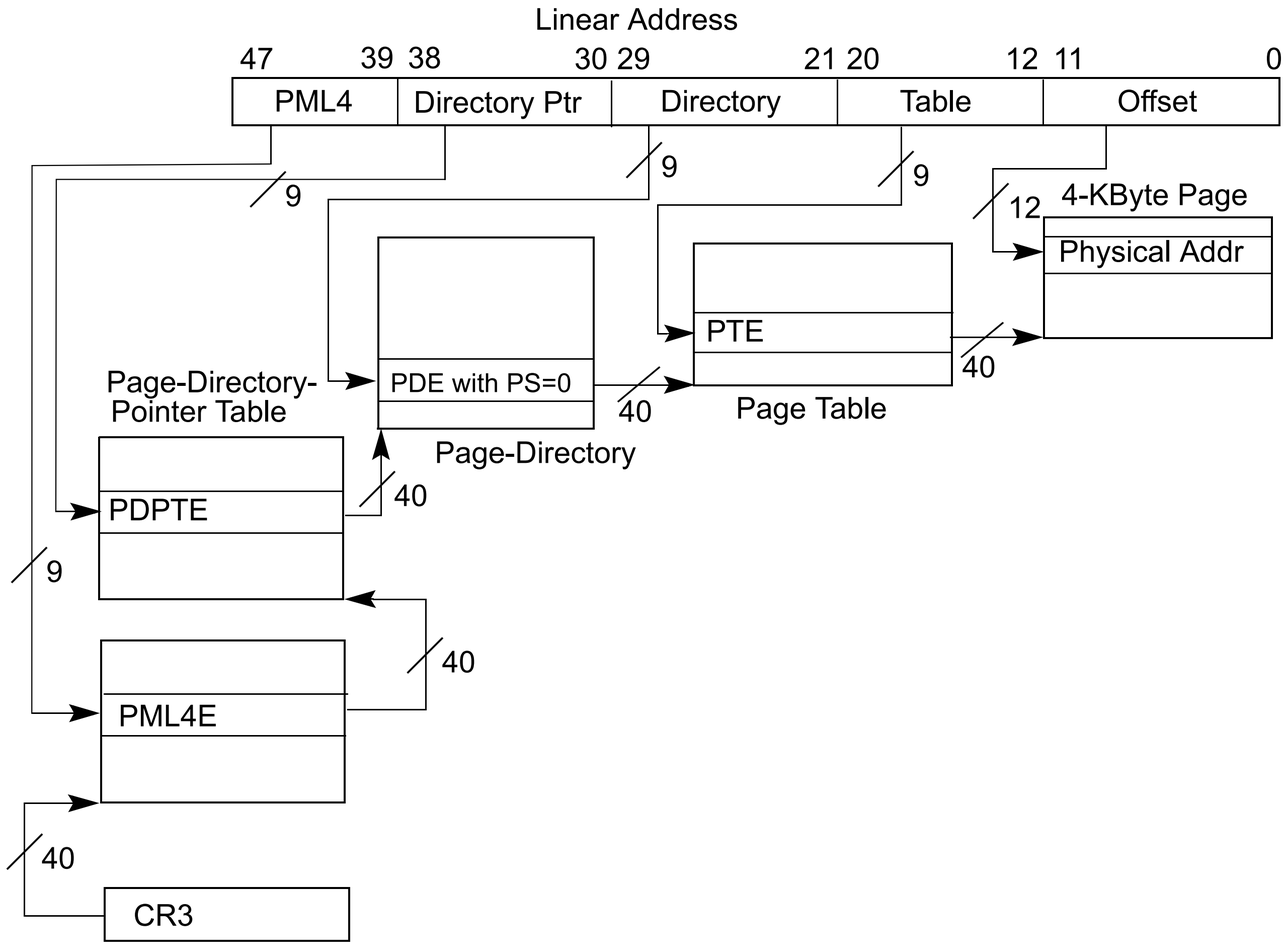
- Long mode: 48-bit virtual addresses (256TB virtual address spaces)
  - 4-levels of paging structures
  - All but two segment registers are forced to a flat model, and no segment limit checking is performed
    - `FS` & `GS` segments can contain non-zero bases (useful for OS)
- Compatibility mode allows 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

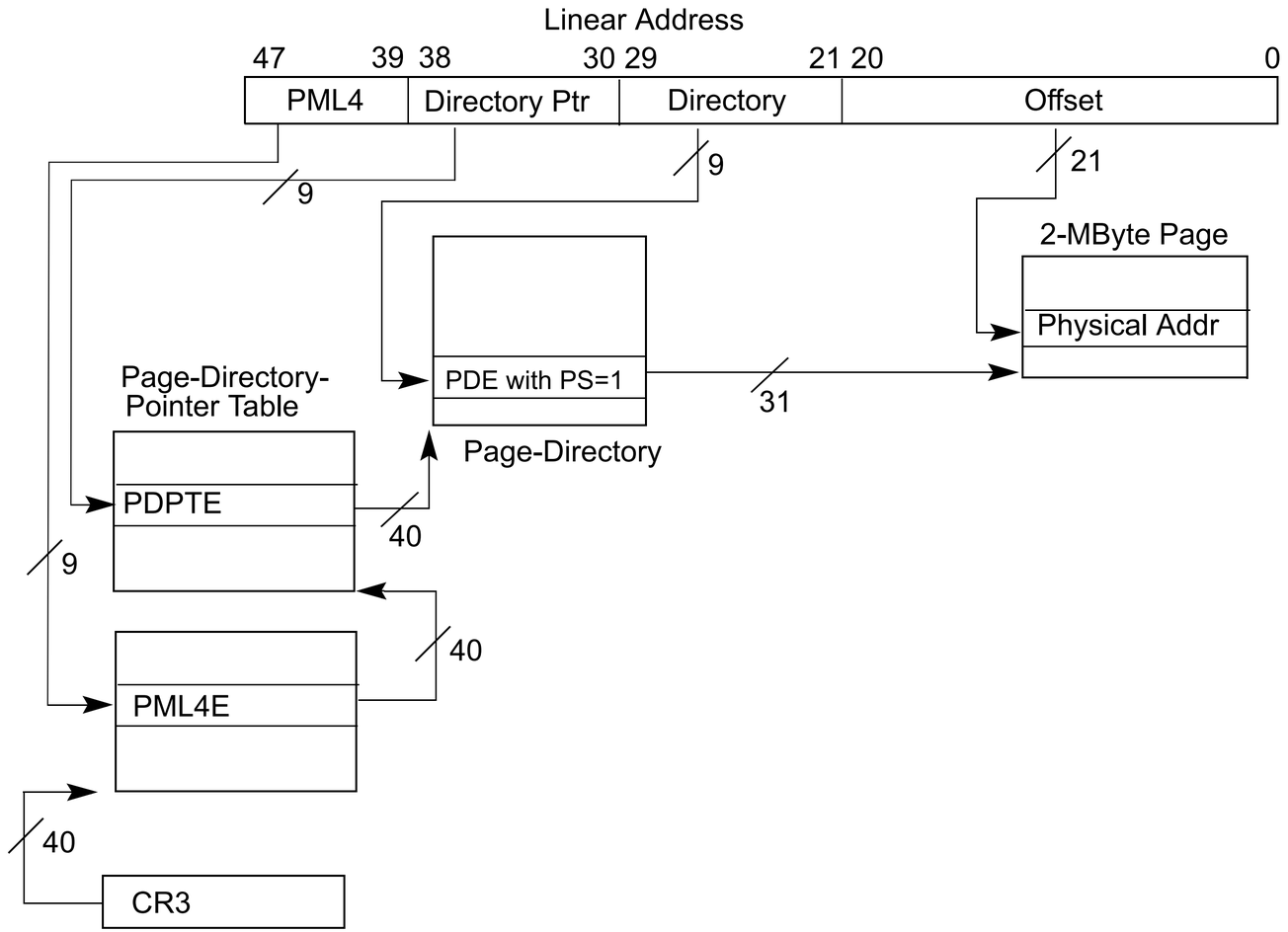
- 48-bit virtual addresses with 4 levels of paging
  - Depending on paging structure entries, supports 4KB, 2MB, 1GB page sizes



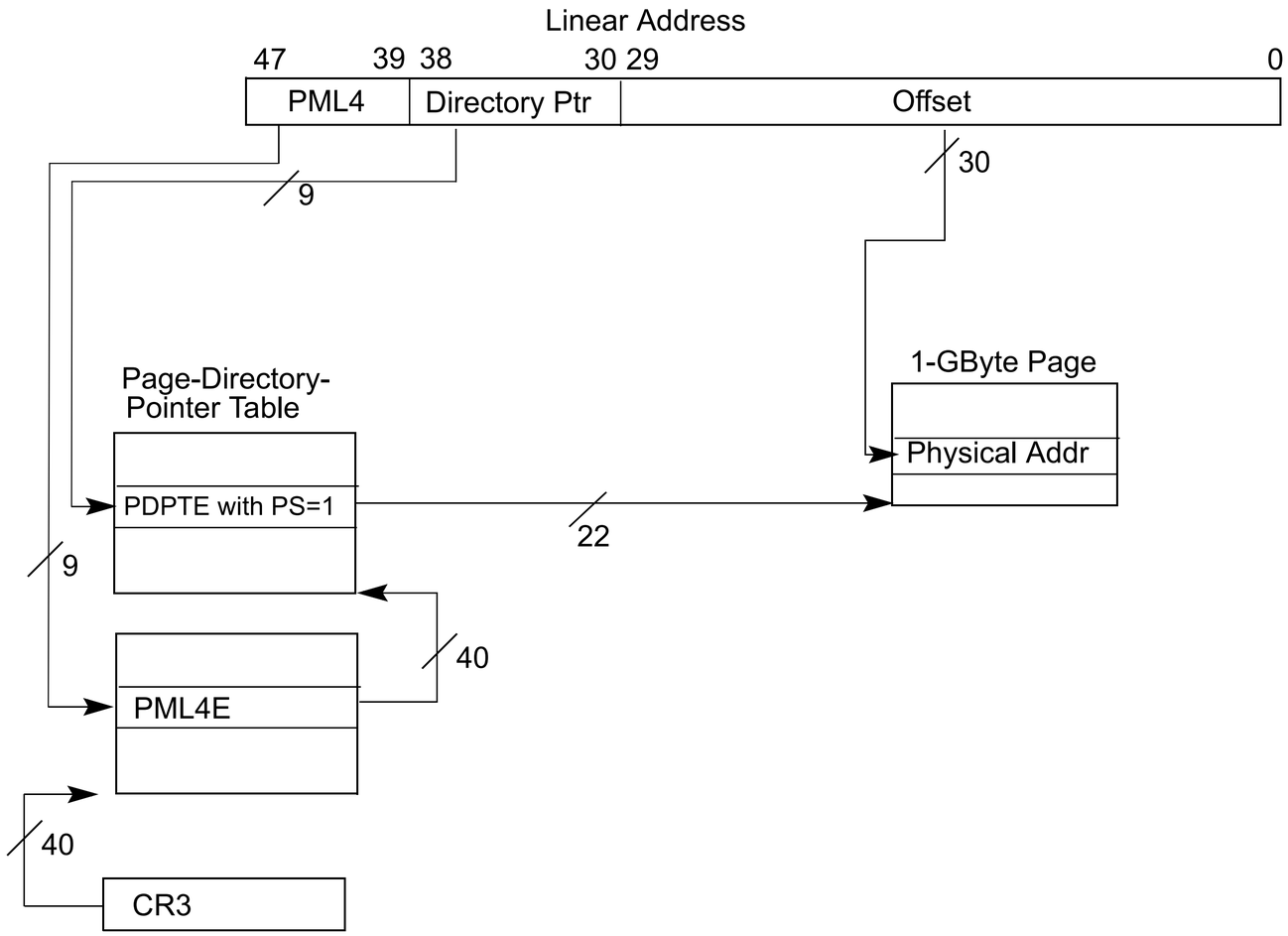
Long mode 4KB paging



Long mode 2MB paging



Long mode 1GB paging



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

63	62	61	60	59	58	57	56	55	54	53	52	51	M[1]	M-1	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved[2]														Address of PML4 table																								Ignored					PCD	PWT	Ign.		CR3
XD	Ignored												Rsvd.	Address of page-directory-pointer table																								Ign.			Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	PML4E: present
Ignored																																															0	PML4E: not present
XD	Prot. Key[4]		Ignored										Rsvd.	Address of 1GB page frame																								Reserved					PAT	Ign.		G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored												Rsvd.	Address of page directory																								Ign.			0	Ign	A	PCD	PWT	U/S	R/W	1	PDPTE: page directory
Ignored																																															0	PDTPE: not present
XD	Prot. Key[4]		Ignored										Rsvd.	Address of 2MB page frame																								Reserved					PAT	Ign.		G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored												Rsvd.	Address of page table																								Ign.			0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																																															0	PDE: not present
XD	Prot. Key[4]		Ignored										Rsvd.	Address of 4KB page frame																								Ign.			PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																																															0	PTE: not present

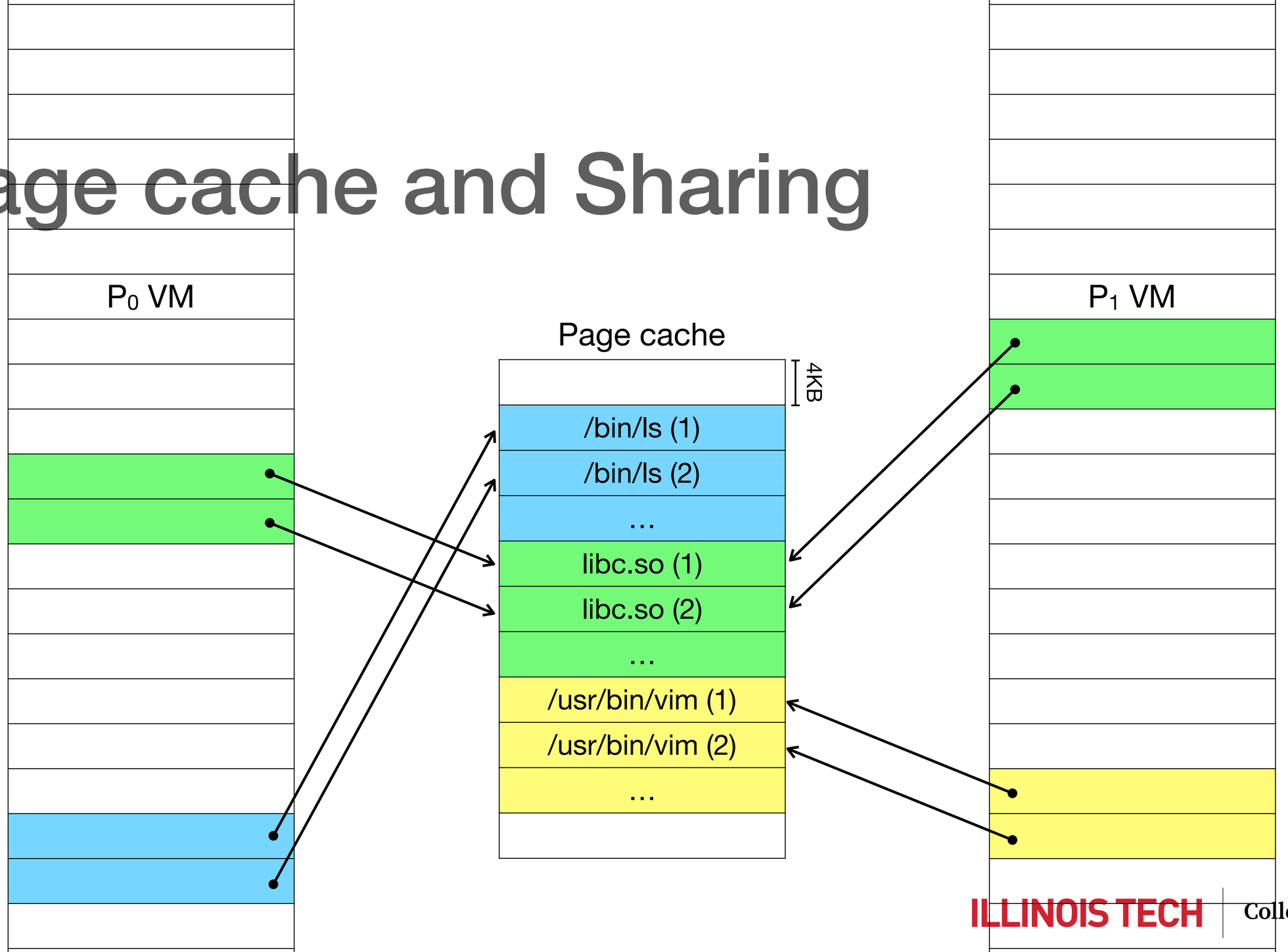
Linux VM features

- 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



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