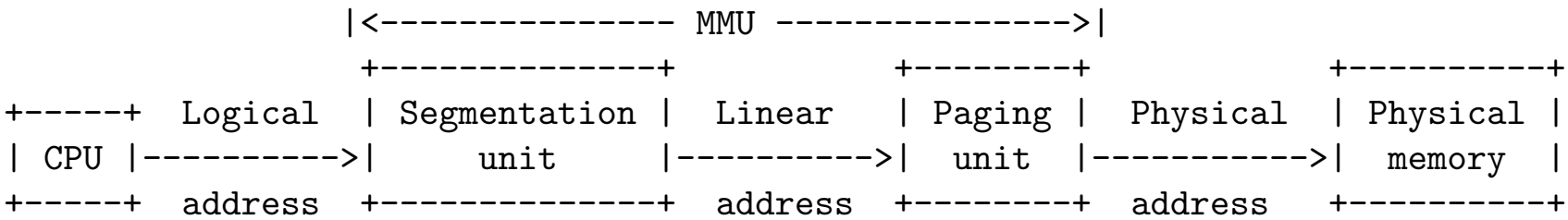
Memory Addressing

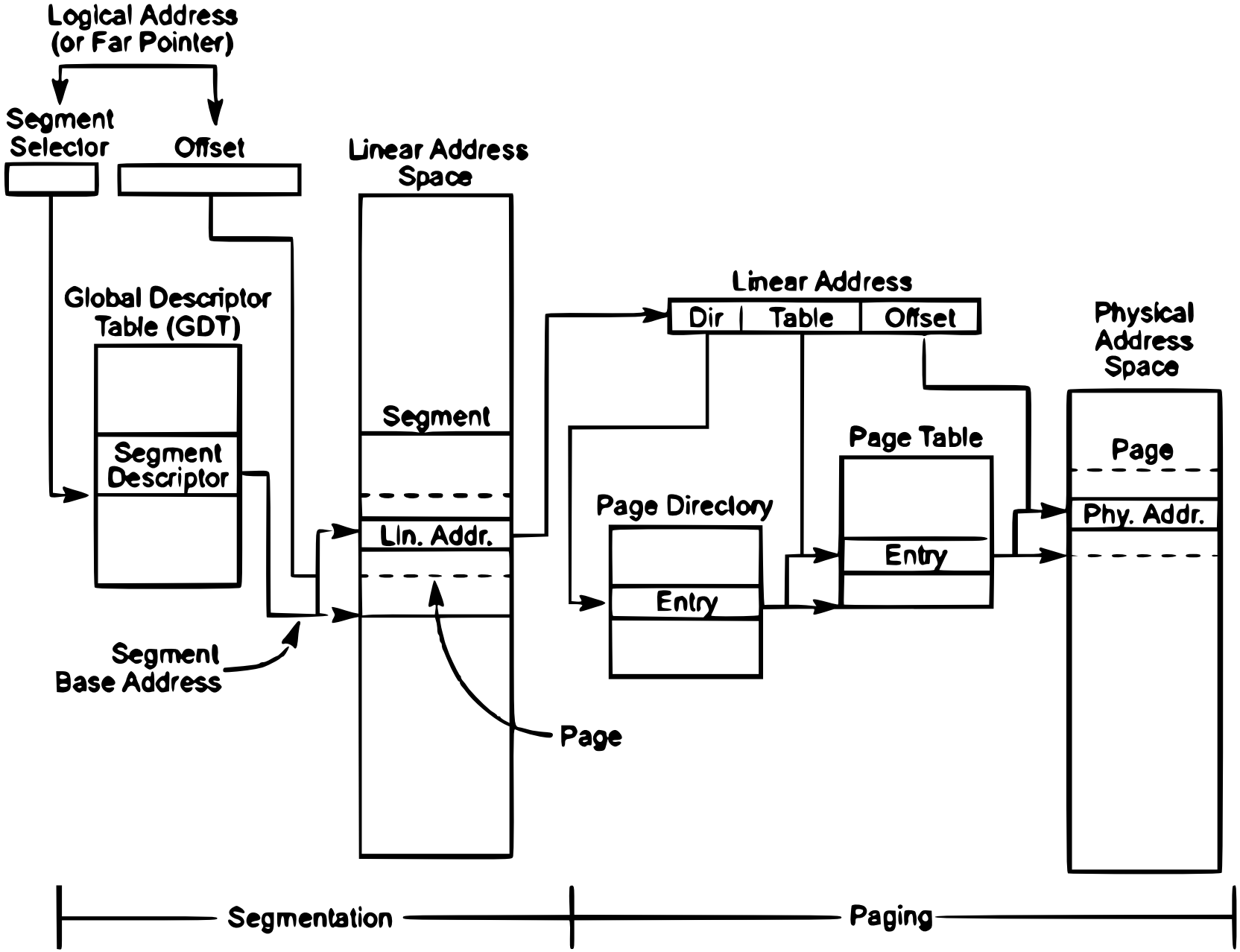
Wang Xiaolin

May 19, 2014

✉ wx672ster+os@gmail.com

Three Kinds Of Addresses




Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

All CPUs Share The Same Memory

Memory Arbiter

if the chip is free
then grants access to a CPU
if the chip is busy servicing a request by another processor
then delay it

Even uniprocessor systems use memory arbiters because of *DMA*.

Real Mode Address Translation

- Backward compatibility of the processors
- BIOS uses real mode addressing
- Use 2 16-bit registers to get a 20-bit address

Logical address format

`<segment:offset>`

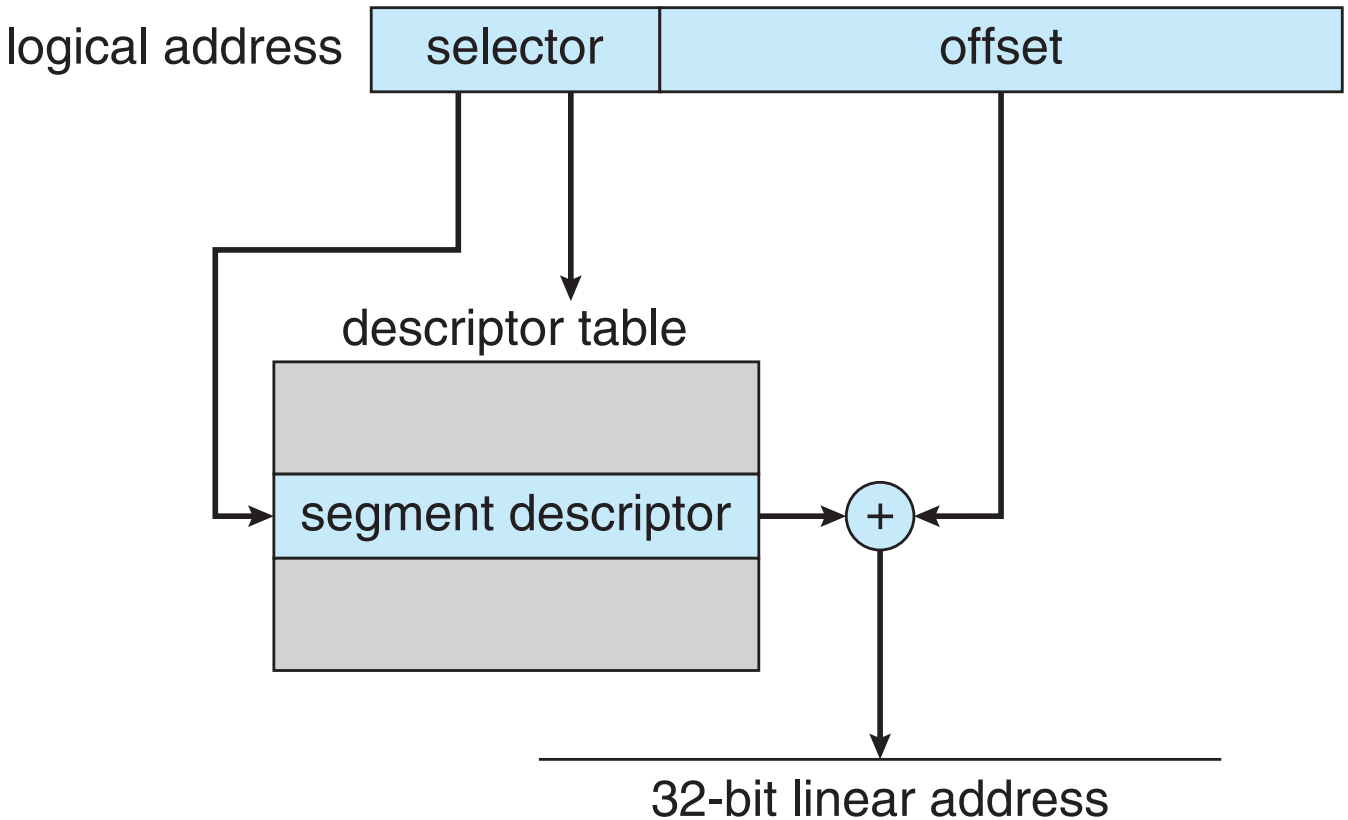
Real mode address translation

$$segment\ number \times 2^4 + offset$$

e.g. to translate <FFFF:0001> into linear address:

$$FFFF \times 16 + 0001 = FFFF0 + 0001 = FFFF1$$

Protected Mode Address Translation



Segment Selectors

A logical address consists of two parts:

segment selector : offset

16 bits 32 bits

Segment selector is an index into GDT/LDT

```
  selector | offset
+-----+-+--+--------+  s - segment number
|  s  |g|p |        |  g - 0-global; 1-local
+-----+-+--+--------+  p - protection use
  13   1 2     32
```

Segmentation Registers

Segment registers hold segment selectors

- **cs** code segment register
  - **CPL** 2-bit, specifies the Current Privilege Level of the CPU
    - 00 - Kernel mode
    - 11 - User mode
- **ss** stack segment register
- **ds** data segment register
- **es/fs/gs** general purpose registers, may refer to arbitrary data segments

Segment Descriptors

All the segments are organized in 2 tables:

GDT *Global Descriptor Table*

- shared by all processes
- GDTR stores address and size of the GDT

LDT *Local Descriptor Table*

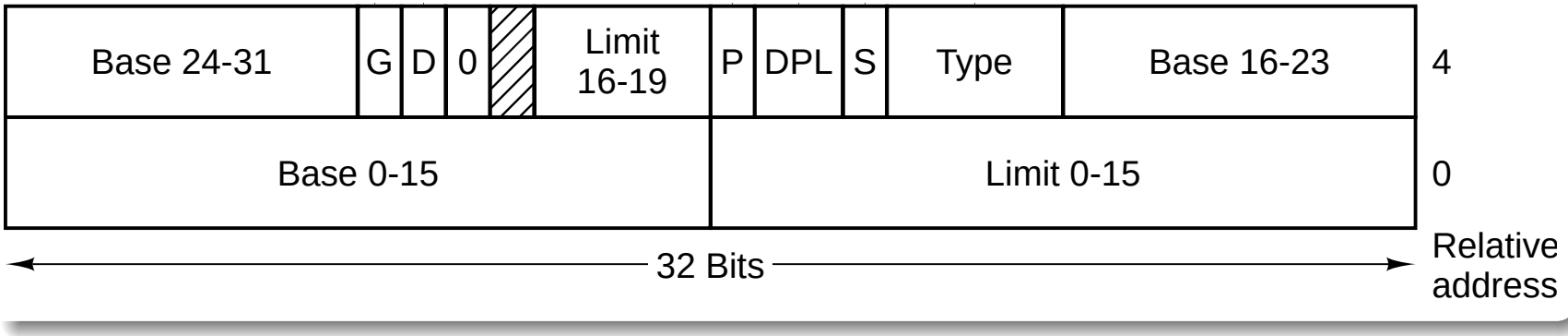
- one process each
- LDTR stores address and size of the LDT

Segment descriptors are entries in either GDT or LDT, 8-byte long

Analogy

Process	$\Longleftrightarrow$	Process Descriptor(PCB)
File	$\Longleftrightarrow$	Inode
Segment	$\Longleftrightarrow$	Segment Descriptor

Example: A LDT entry for code segment



Base: Where the segment starts

Limit: 20 bit, $\Rightarrow 2^{20}$ in size

G: Granularity flag

- 0 - segment size in bytes
- 1 - in 4096 bytes

S: System flag

- 0 - system segment, e.g. LDT
- 1 - normal code/data segment

D/B:
- 0 - 16-bit offset
- 1 - 32-bit offset

Type: segment type (cs/ds/tss)

TSS: Task status, i.e. it's executing or not

DPL: Descriptor Privilege Level. 0 or 3

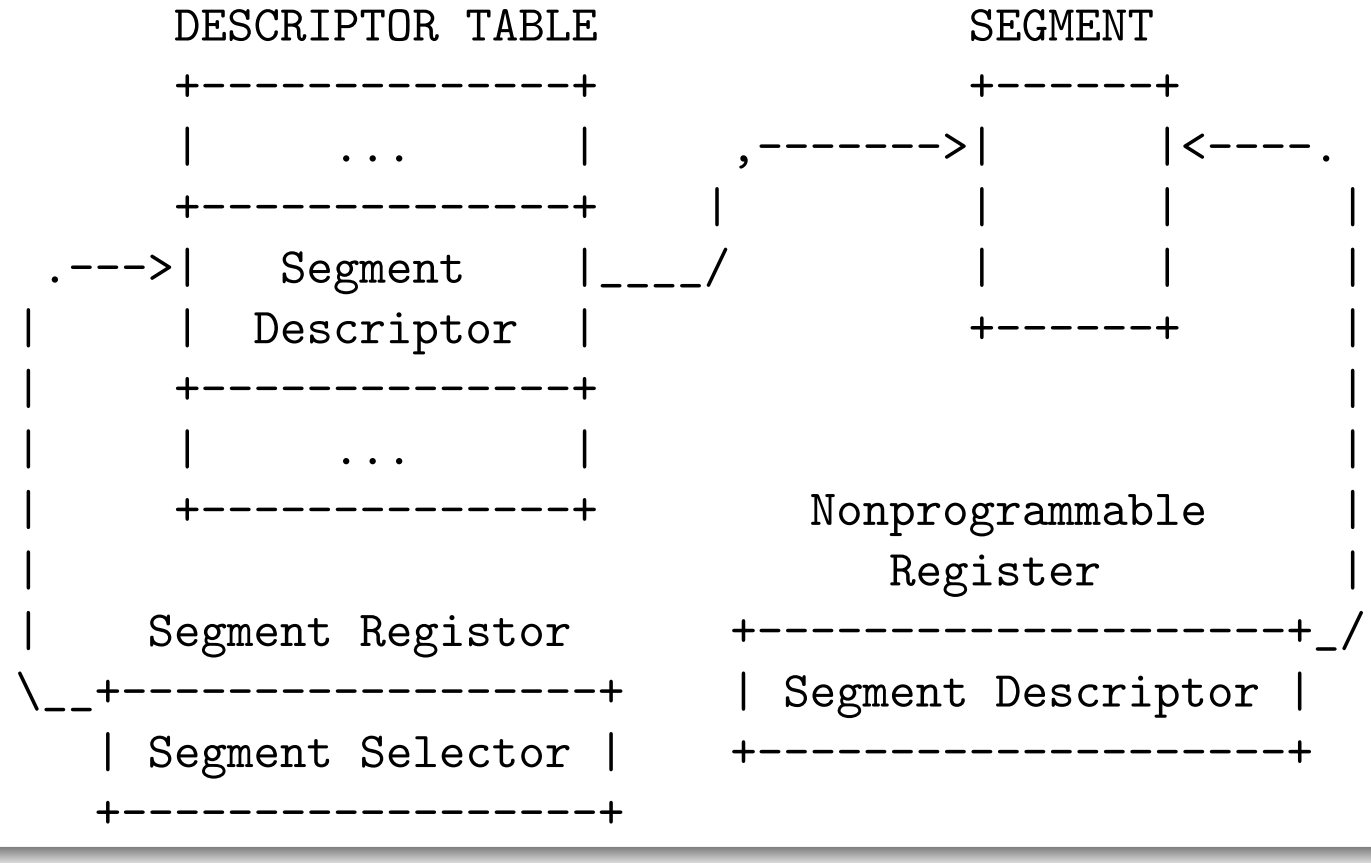
P: Segment-Present flag

- 0 - not in memory
- 1 - in memory

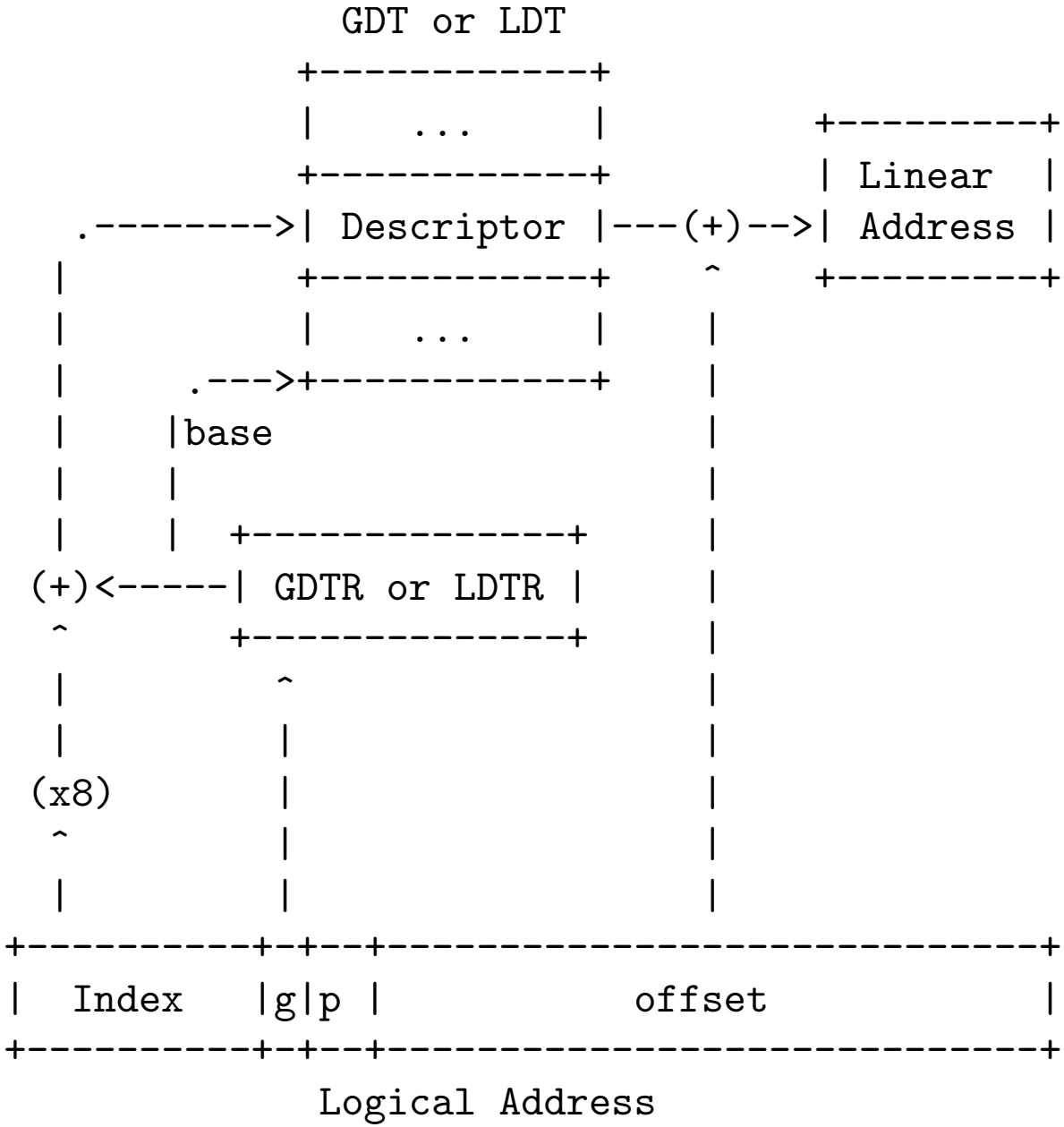
AVL: ignored by Linux

Fast Access to Segment Descriptors

a non-programmable cache register for each segment register



Translating a logical address



Linux prefers paging to segmentation

Because

- Segmentation and paging are somewhat redundant
- Memory management is simpler when all processes share the same set of linear addresses
- Maximum portability. RISC architectures in particular have limited support for segmentation

The Linux 2.6 uses segmentation only when required by the 80x86 architecture.

The Linux GDT Layout

Each GDT includes 18 segment descriptors and 14 null, unused, or reserved entries

include/asm-i386/segment.h

0	null
1	reserved
2	reserved
3	reserved
4	unused
5	unused
6	TLS segment #1
7	TLS segment #2
8	TLS segment #3
9	reserved
10	reserved
11	reserved
12	kernel code segment
13	kernel data segment
14	default user CS
15	default user DS
16	TSS
17	LDT
18	PNPBIOS support
19	PNPBIOS support
20	PNPBIOS support
21	PNPBIOS support
22	PNPBIOS support
23	APM BIOS support
24	APM BIOS support
25	APM BIOS support
26	ESPFIX small SS
27	per-cpu
28	stack_canary-20
29	unused
30	unused
31	TSS for double fault handler

The Four Main Linux Segments

Every process in Linux has these 4 segments

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xfffff	1	10	3	1	1
user data	0x00000000	1	0xfffff	1	2	3	1	1
kernel code	0x00000000	1	0xfffff	1	10	0	1	1
kernel data	0x00000000	1	0xfffff	1	2	0	1	1

All linear addresses start at 0, end at 4G-1

- All processes share the same set of linear addresses
- Logical addresses coincide with linear addresses

Segment Selectors

include/asm-i386/segment.h

```
#define GDT_ENTRY_DEFAULT_USER_CS       14
#define __USER_CS (GDT_ENTRY_DEFAULT_USER_CS * 8 + 3)

#define GDT_ENTRY_DEFAULT_USER_DS       15
#define __USER_DS (GDT_ENTRY_DEFAULT_USER_DS * 8 + 3)

#define GDT_ENTRY_KERNEL_BASE   12

#define GDT_ENTRY_KERNEL_CS             (GDT_ENTRY_KERNEL_BASE + 0)
#define __KERNEL_CS (GDT_ENTRY_KERNEL_CS * 8)

#define GDT_ENTRY_KERNEL_DS             (GDT_ENTRY_KERNEL_BASE + 1)
#define __KERNEL_DS (GDT_ENTRY_KERNEL_DS * 8)
```

$Selector = Index \ll 3 + G + RPL$

__USER_CS	$14 \ll 3 + 3 = 115$	0000 0000 0111 0011
__USER_DS	$15 \ll 3 + 3 = 123$	0000 0000 0111 1011
__KERNEL_CS	$12 \ll 3 + 0 = 96$	0000 0000 0110 0000
__KERNEL_DS	$13 \ll 3 + 0 = 104$	0000 0000 0110 1000

Example:

To address the kernel code segment, the kernel just loads the value yielded by the `__KERNEL_CS` macro into the `cs` segmentation register.

Note that

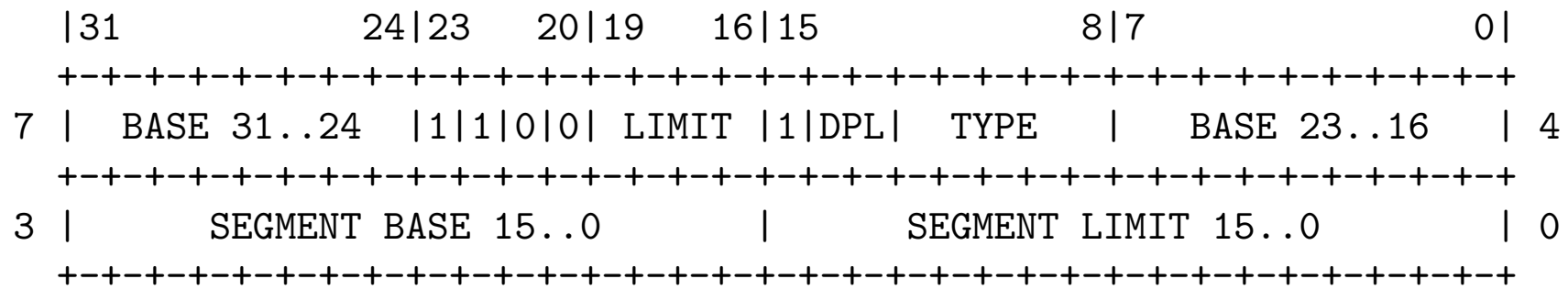
1. `base = 0`
2. `limit = 0xfffff`

This means that

- all processes, either in User Mode or in Kernel Mode, may use the same logical addresses
- logical addresses (Offset fields) coincide with linear addresses

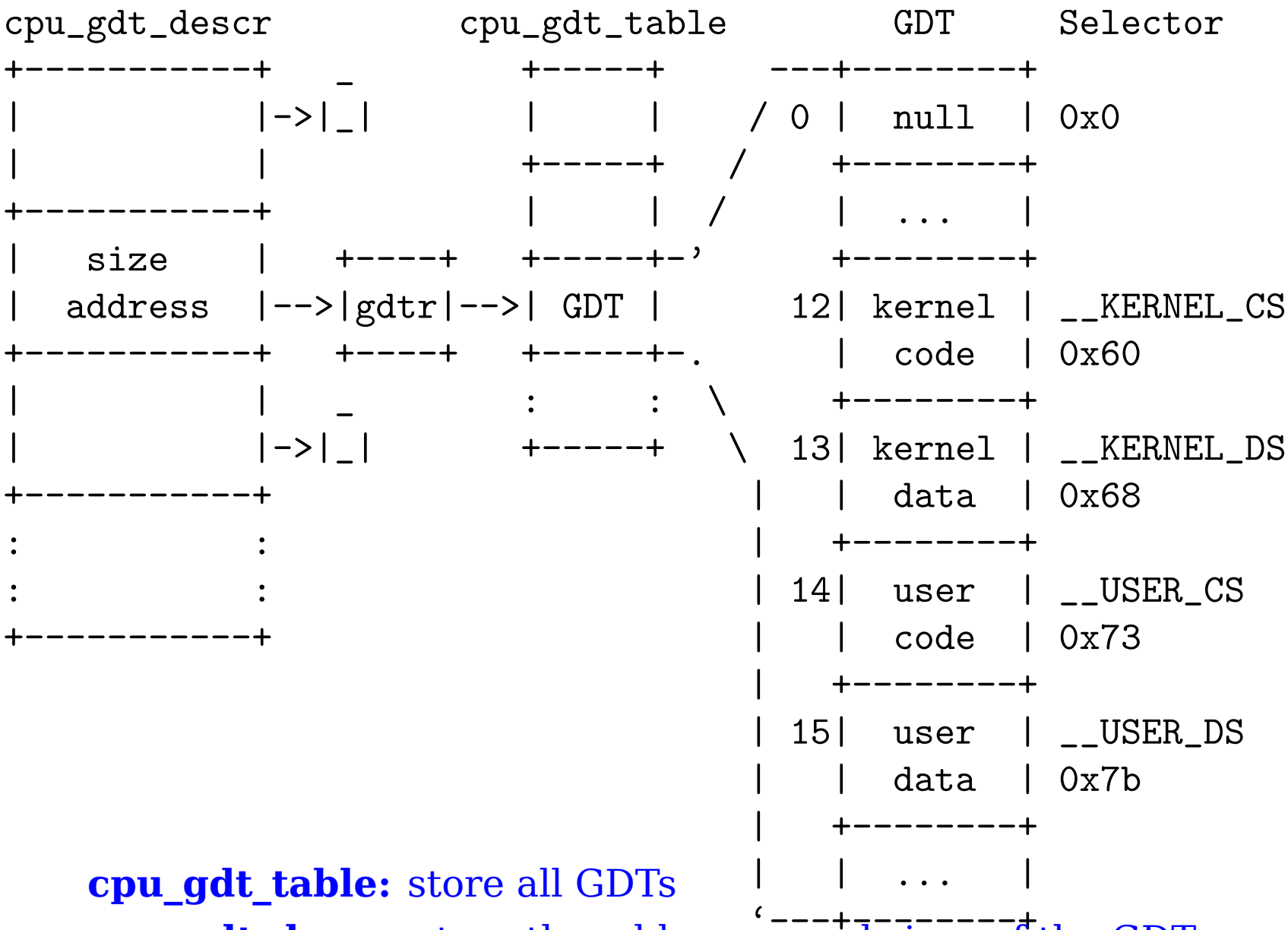
The Linux GDT

— 8 byte segment descriptor



arch/i386/kernel/head.S

```
ENTRY(cpu_gdt_table)
  .quad 0x00cf9a000000ffff /* 0x60 kernel 4GB code at 0x00000000 */
  .quad 0x00cf92000000ffff /* 0x68 kernel 4GB data at 0x00000000 */
  .quad 0x00cffa000000ffff /* 0x73 user 4GB code at 0x00000000 */
  .quad 0x00cff2000000ffff /* 0x7b user 4GB data at 0x00000000 */
```



cpu_gdt_table: store all GDTs

cpu_gdt_descr: store the addresses and sizes of the GDTs

Paging in Hardware

— Starting with the 80386, all 80x86 processors support paging

A page is

- a set of linear addresses
- a block of data

A page frame is

- a constituent of main memory
- a storage area

A page table

- is a data structure
- maps linear to physical addresses
- stored in main memory

Pentium Paging

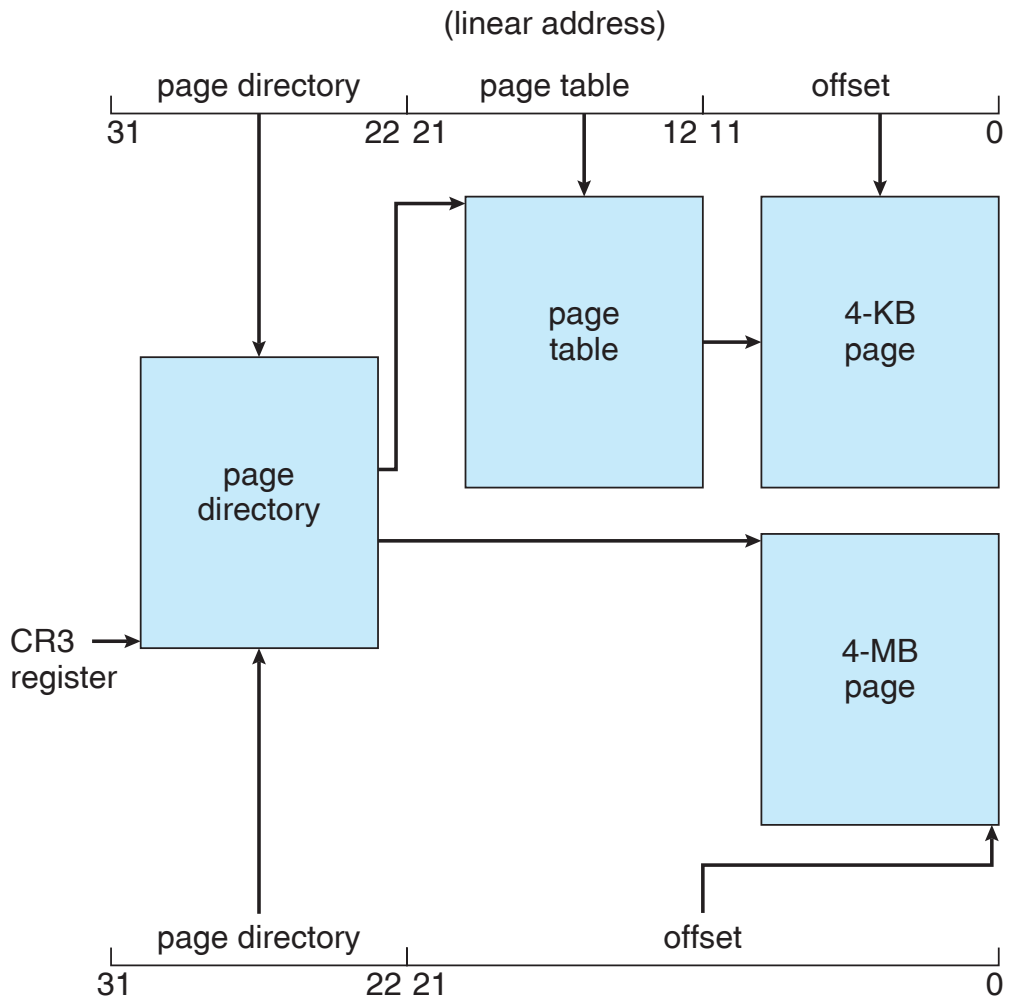
— Linear Address ⇒ Physical Address

Two page size in Pentium:

4K: 2-level paging

4M: 1-level paging

```
      page number      | page offset
+----------+----------+------------+
|    p1    |    p2    |     d      |
+----------+----------+------------+
    10          10           12
     |           |
     |           `-> pointing to 1k frames
     `--> pointing to 1k page tables
```



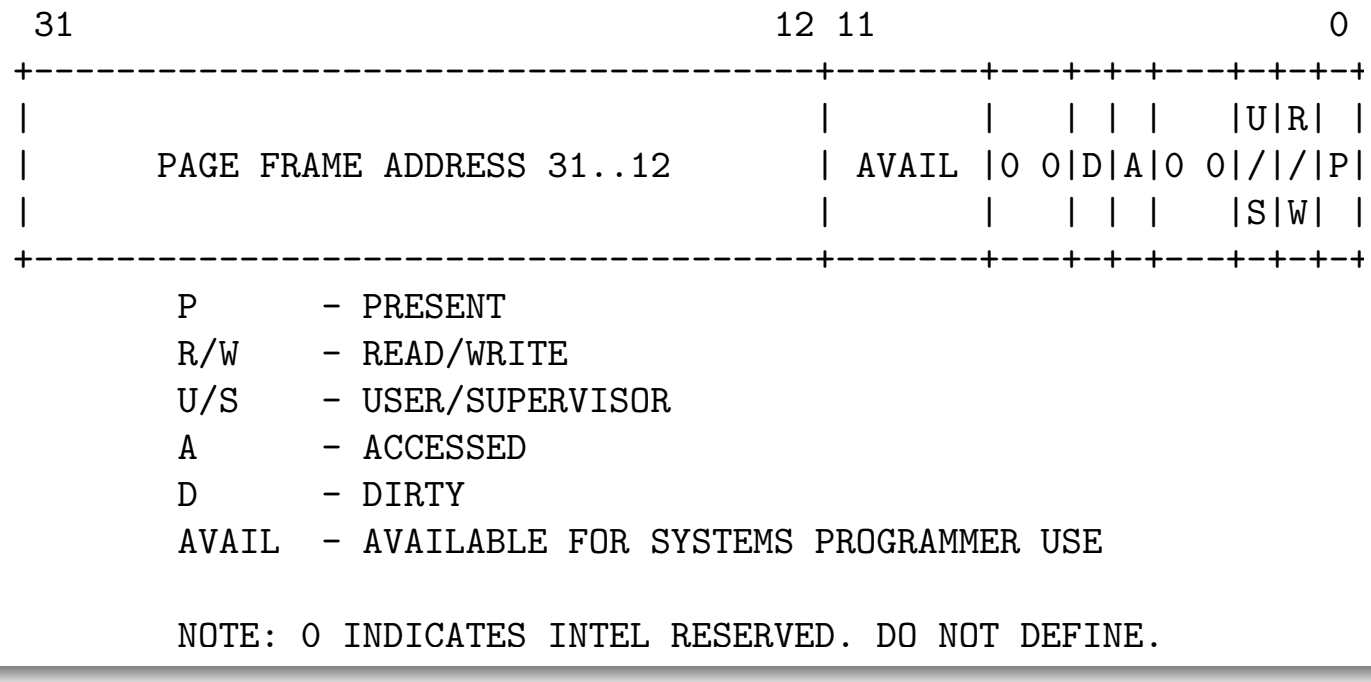
Same structure for Page Dirs and Page Tables

- 4 bytes (32 bits) long
- Page size is usually 4k (2^{12} bytes). OS dependent

  ~$ `getconf PAGESIZE`
- Could have $2^{32-12} = 2^{20} = 1M$ pages

  Could addressing $1M \times 4KB = 4GB$ memory

Intel i386 page table entry



Physical Address Extension (PAE)

— 32-bit linear⇒ 36-bit physical

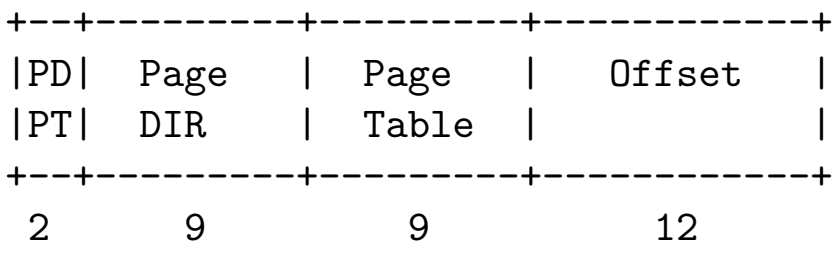
Need a new paging mechanism

	Linear Address	Physical Address	Max RAM	Page Size	PTE Size	Paging Level
No PAE	32 bits	32 bits	$2^{32} = 4GB$	4K,4M	32 bits	1,2
PAE	32 bits	36 bits	$2^{36} = 64GB$	4K,2M	64 bits	2,3

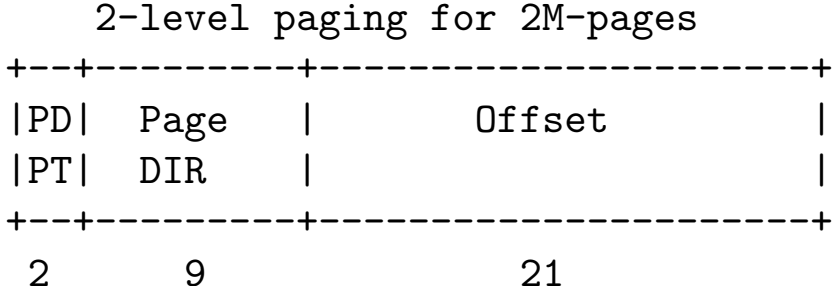
PDPT Page Directory Pointer Table, is a new level of Page Table

`64-bit entry` $\times 4$

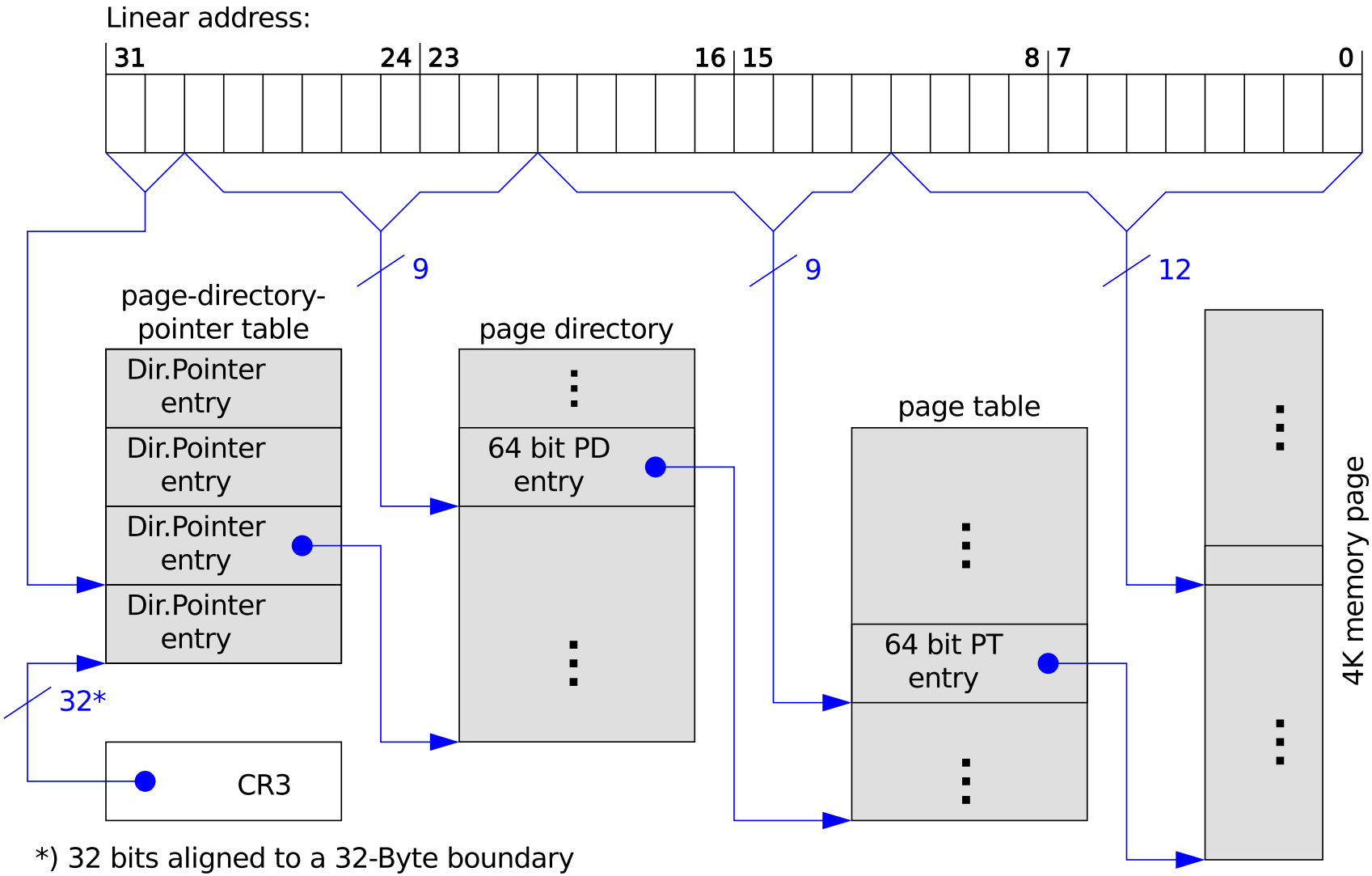
3-level paging for 4K-pages



2-level paging for 2M-pages

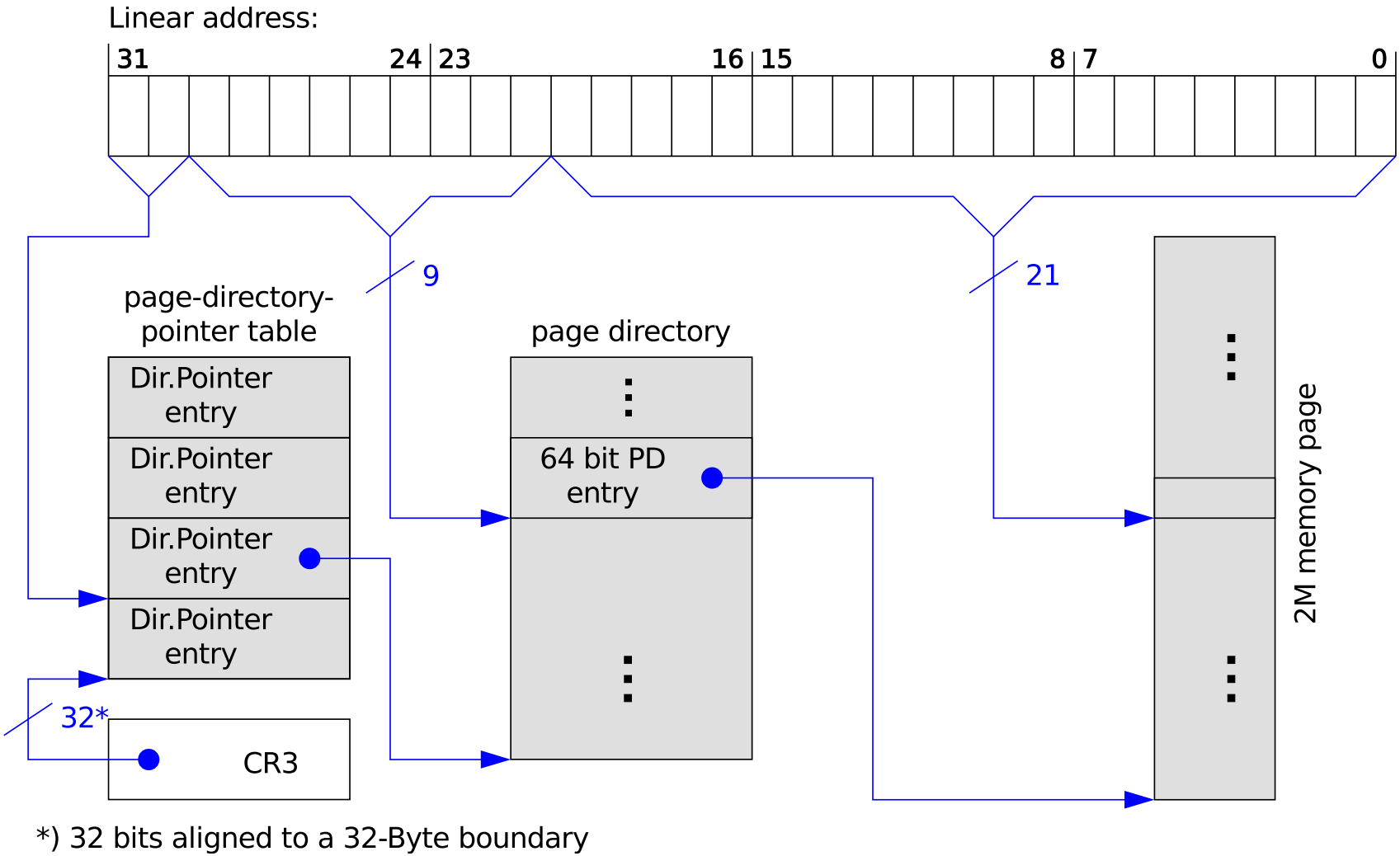


PAE with 4K pages



*) 32 bits aligned to a 32-Byte boundary

PAE with 2M pages



*) 32 bits aligned to a 32-Byte boundary

Physical Address Extension (PAE)

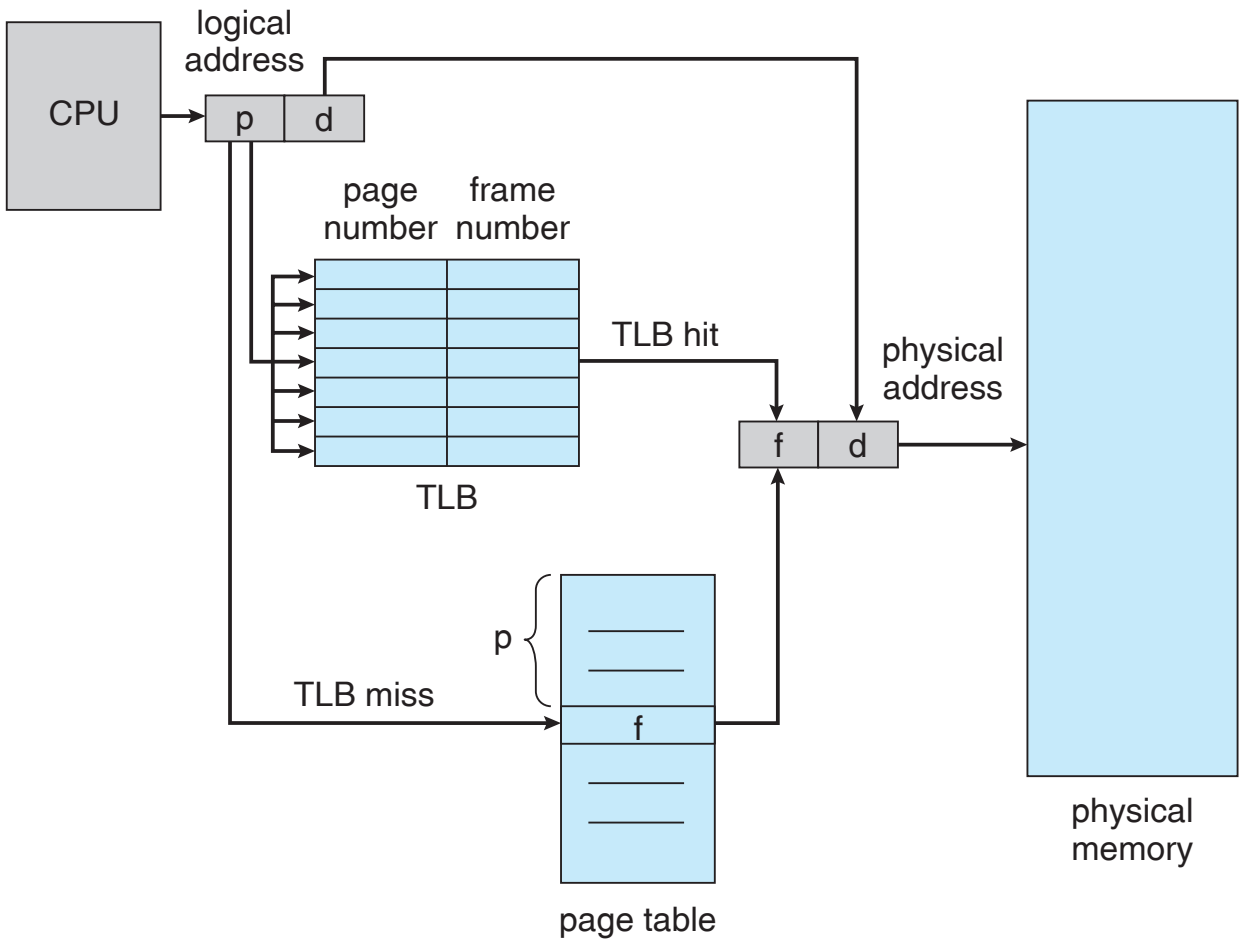
The linear address are still 32 bits

- A process cannot use more than 4G RAM
- The kernel programmers have to reuse the same linear addresses to map 64GB RAM
- The number of processes is increased

Translation Lookaside Buffers (TLB)

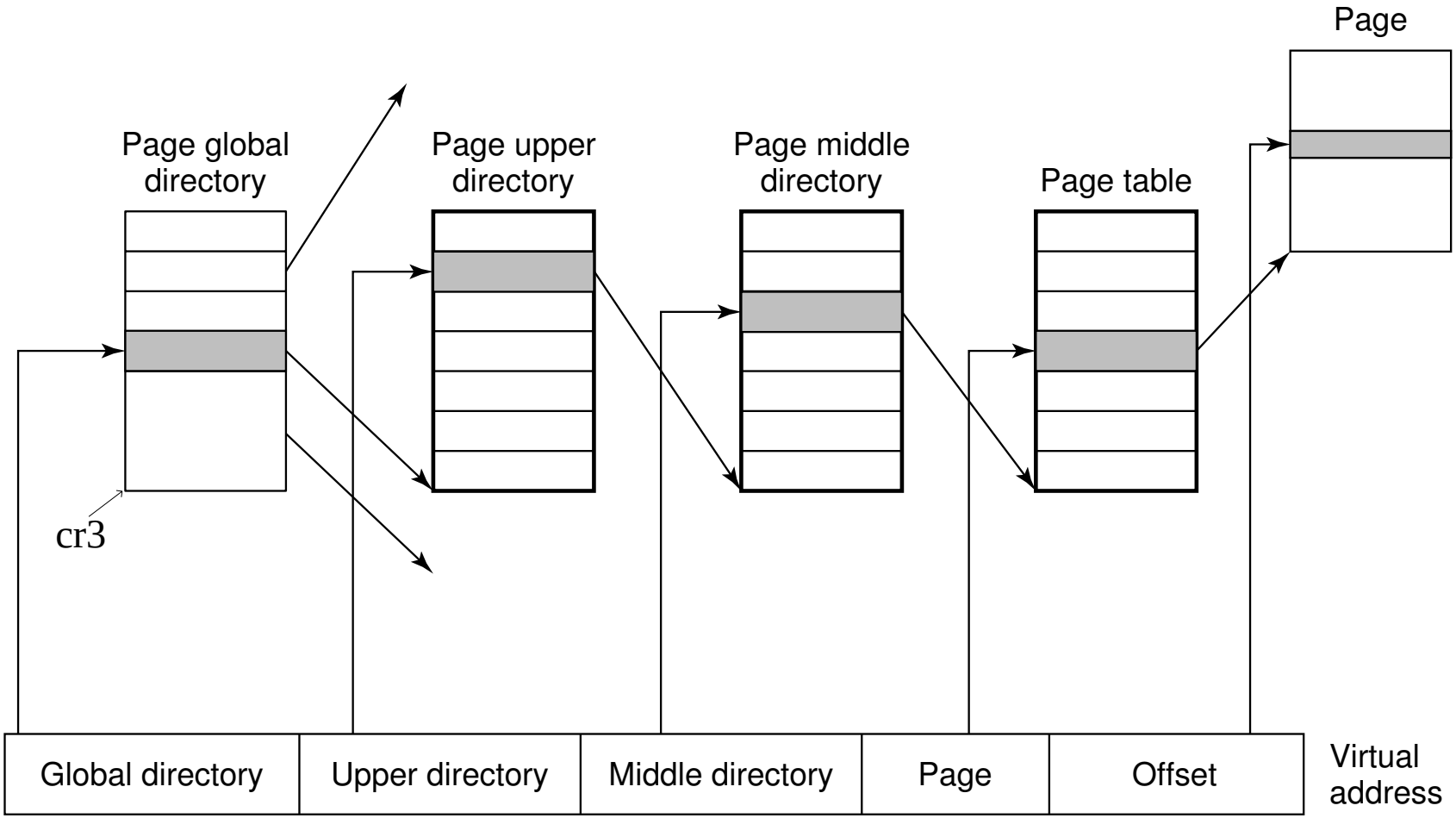
Fact: 80-20 rule

- Only a small fraction of the PTEs are heavily read; the rest are barely used at all



Paging In Linux

— 4-level paging for both 32-bit and 64-bit



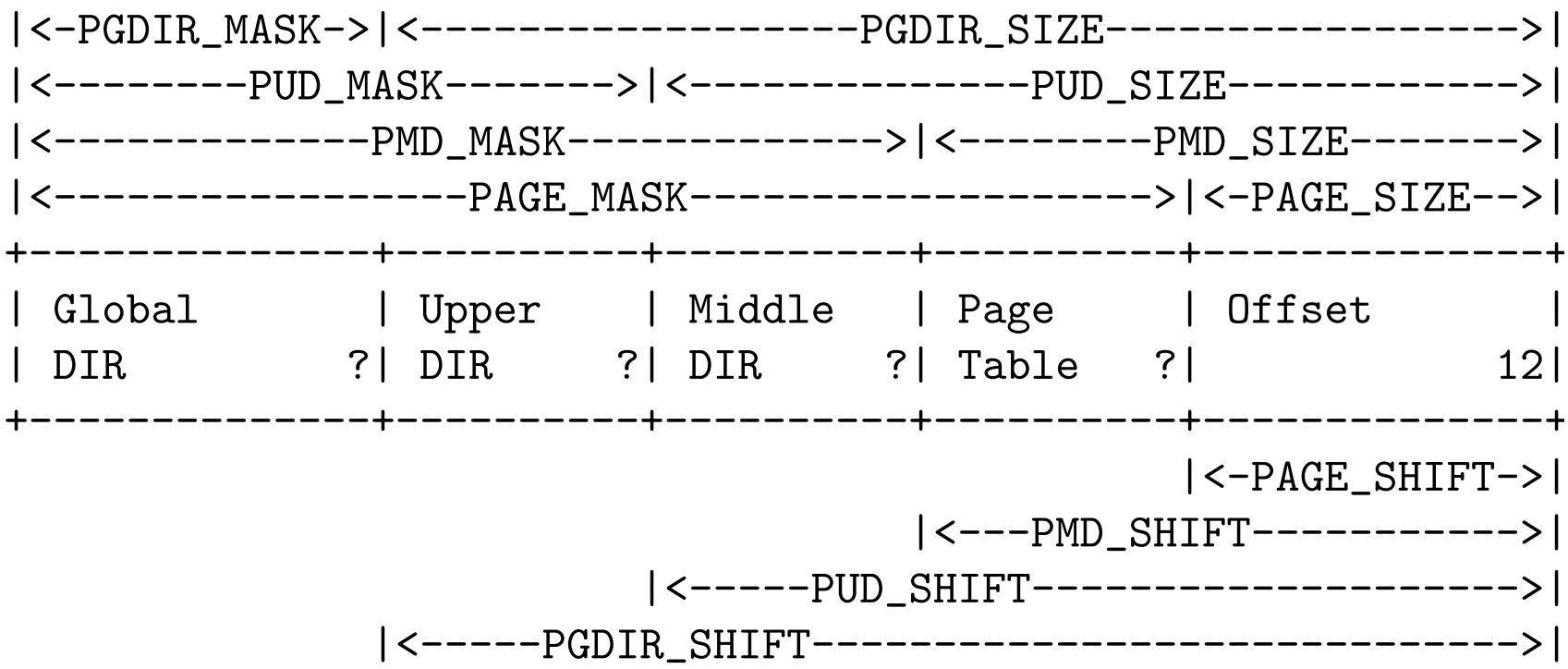
4-level paging for both 32-bit and 64-bit

- 64-bit: four-level paging
  1. Page Global Directory
  2. Page Upper Directory
  3. Page Middle Directory
  4. Page Table
- 32-bit: two-level paging
  1. Page Global Directory
  2. Page Upper Directory — 0 bits; 1 entry
  3. Page Middle Directory — 0 bits; 1 entry
  4. Page Table

The same code can work on 32-bit and 64-bit architectures

Arch	Page size	Address bits	Paging levels	Address splitting
x86	4KB(12bits)	32	2	$10+0+0+10+12$
x86-PAE	4KB(12bits)	32	3	$2+0+9+9+12$
x86-64	4KB(12bits)	48	4	$9+9+9+9+12$

The Linear Address Fields



- `*_SHIFT` to specify the number of bits being mapped
- `*_MASK` to mask out all the upper bits
- `*_SIZE` how many bytes are addressed by each entry

`*_MASK` and `*_SIZE` values are calculated based on `*_SHIFT`

include/asm-i386/page.h

```
/* PAGE_SHIFT determines the page size */
#define PAGE_SHIFT      12
#define PAGE_SIZE       (1UL << PAGE_SHIFT)
#define PAGE_MASK       (~(PAGE_SIZE-1))

#define LARGE_PAGE_MASK (~(LARGE_PAGE_SIZE-1))
#define LARGE_PAGE_SIZE (1UL << PMD_SHIFT)
```

PAGE_SIZE: $2^{12} = 4k$

PAGE_MASK: 0xfffff000

LARGE_PAGE_SIZE: depends

PAE: $2^{21} = 2M$

no PAE: $2^{22} = 4M$

Compile Time Dual-mode

include/asm-i386/pgtable.h

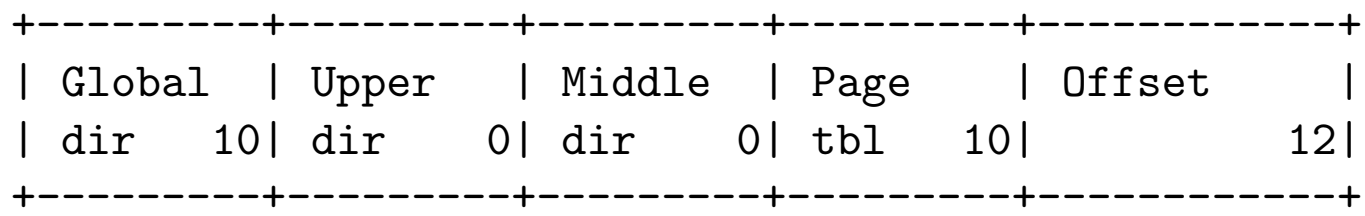
```
/*
 * The Linux x86 paging architecture is 'compile-time dual-mode', it
 * implements both the traditional 2-level x86 page tables and the
 * newer 3-level PAE-mode page tables.
 */
#ifdef CONFIG_X86_PAE
# include <asm/pgtable-3level_types.h>
# define PMD_SIZE       (1UL << PMD_SHIFT)
# define PMD_MASK       (~(PMD_SIZE - 1))
#else
# include <asm/pgtable-2level_types.h>
#endif

#define PGDIR_SIZE      (1UL << PGDIR_SHIFT)
#define PGDIR_MASK      (~(PGDIR_SIZE - 1))
```

	PMD_SHIFT	PUD_SHIFT	PGDIR_SHIFT
2-level	22	22	22
3-level	21	21	30

```
include/asm-i386/pgtable-2level-defs.h #define PGDIR_SHIFT      22
include/asm-i386/pgtable-3level-defs.h #define PGDIR_SHIFT      30
include/asm-x86_64/pgtable.h #define PGDIR_SHIFT  39
```

2-level — no PAE, 4K-page

PMD and PUD are folded



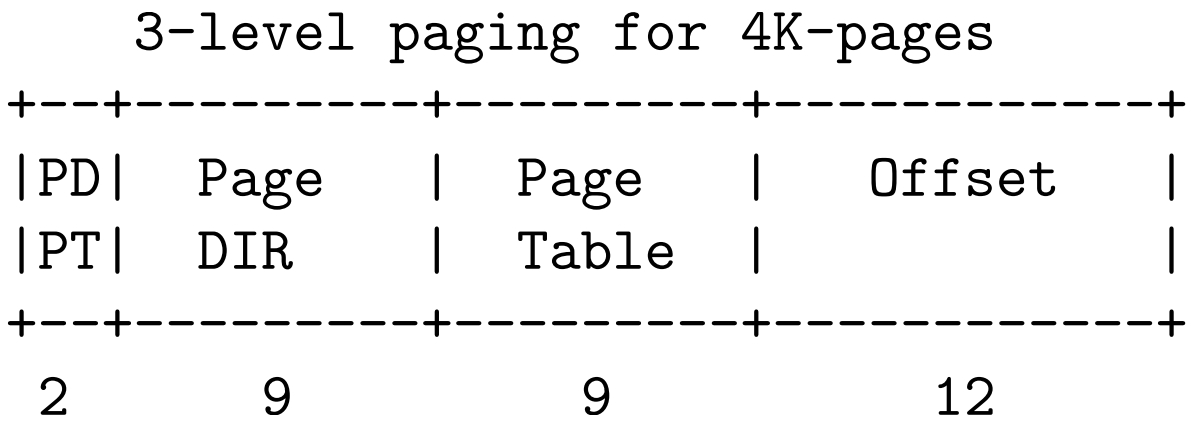
include/asm-generic/pgtable-nopud.h

```
#define PUD_SHIFT       PGDIR_SHIFT
#define PTRS_PER_PUD    1
#define PUD_SIZE        (1UL << PUD_SHIFT)
#define PUD_MASK        (~(PUD_SIZE-1))
```

include/asm-generic/pgtable-nopmd.h

```
#define PMD_SHIFT       PUD_SHIFT
#define PTRS_PER_PMD    1
#define PMD_SIZE        (1UL << PMD_SHIFT)
#define PMD_MASK        (~(PMD_SIZE-1))
```

3-level — PAE enabled



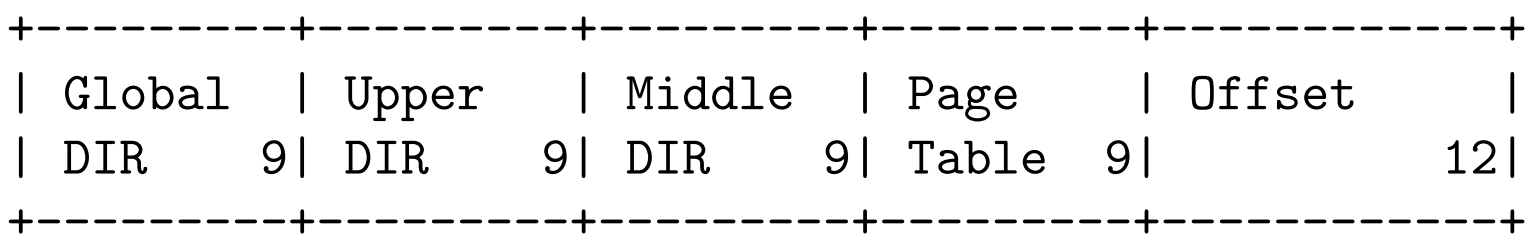
include/asm-i386/pgtable-3level-defs.h

```
#define PGDIR_SHIFT      30
#define PTRS_PER_PGD     4
#define PMD_SHIFT        21
#define PTRS_PER_PMD     512
```

PUD is eliminated

4-level — x86_64

48 address bits



include/asm-x86_64/pgtable.h

```
#define PGDIR_SHIFT     39
#define PTRS_PER_PGD    512

#define PUD_SHIFT       30
#define PTRS_PER_PUD    512

#define PMD_SHIFT       21
#define PTRS_PER_PMD    512
```

Page Table Handling

— Data formats

include/asm-i386/page.h

```
#ifdef CONFIG_X86_PAE
extern unsigned long long __supported_pte_mask;
typedef struct { unsigned long pte_low, pte_high; } pte_t;
typedef struct { unsigned long long pmd; } pmd_t;
typedef struct { unsigned long long pgd; } pgd_t;
typedef struct { unsigned long long pgprot; } pgprot_t;
#define pmd_val(x)      ((x).pmd)
#define pte_val(x)      ((x).pte_low | ((unsigned long long)(x).pte_high << 32))
#define __pmd(x) ((pmd_t) { (x) } )
#define HPAGE_SHIFT     21
#else
typedef struct { unsigned long pte_low; } pte_t;
typedef struct { unsigned long pgd; } pgd_t;
typedef struct { unsigned long pgprot; } pgprot_t;
#define boot_pte_t pte_t /* or would you rather have a typedef */
#define pte_val(x)      ((x).pte_low)
#define HPAGE_SHIFT     22
#endif
```

Page Table Handling

— Read or modify page table entries

Macros and functions

pte_none	pte_clear	set_pte	pte_same(a,b)
pte_present	pte_user()	pte_read()	pte_write()
pte_exec()	pte_dirty()	pte_young()	pte_file()
mk_pte_huge()	pte_wrprotect()	pte_rdprotect()	pte_exprotect()
pte_mkwrite()	pte_mkread()	pte_mkexec()	pte_mkclean()
pte_mkdirty()	pte_mkold()	pte_mkyoung()	pte_modify(p,v)
mk_pte(p,prot)	pte_index(addr)	pte_page(x)	pte_to_pgoff(pte)

a lot more for pmd, pud, pgd ...

Example — To find a page table entry

mm/memory.c

```
pgd_t *pgd;
pud_t *pud;
pmd_t *pmd;
pte_t *ptep, pte;

pgd = pgd_offset(mm, address);
if (pgd_none(*pgd) || unlikely(pgd_bad(*pgd)))
  goto out;

pud = pud_offset(pgd, address);
if (pud_none(*pud) || unlikely(pud_bad(*pud)))
  goto out;

pmd = pmd_offset(pud, address);
if (pmd_none(*pmd) || unlikely(pmd_bad(*pmd)))
  goto out;

ptep = pte_offset_map(pmd, address);
if (!ptep)
  goto out;

pte = *ptep;
```

Physical Memory Layout

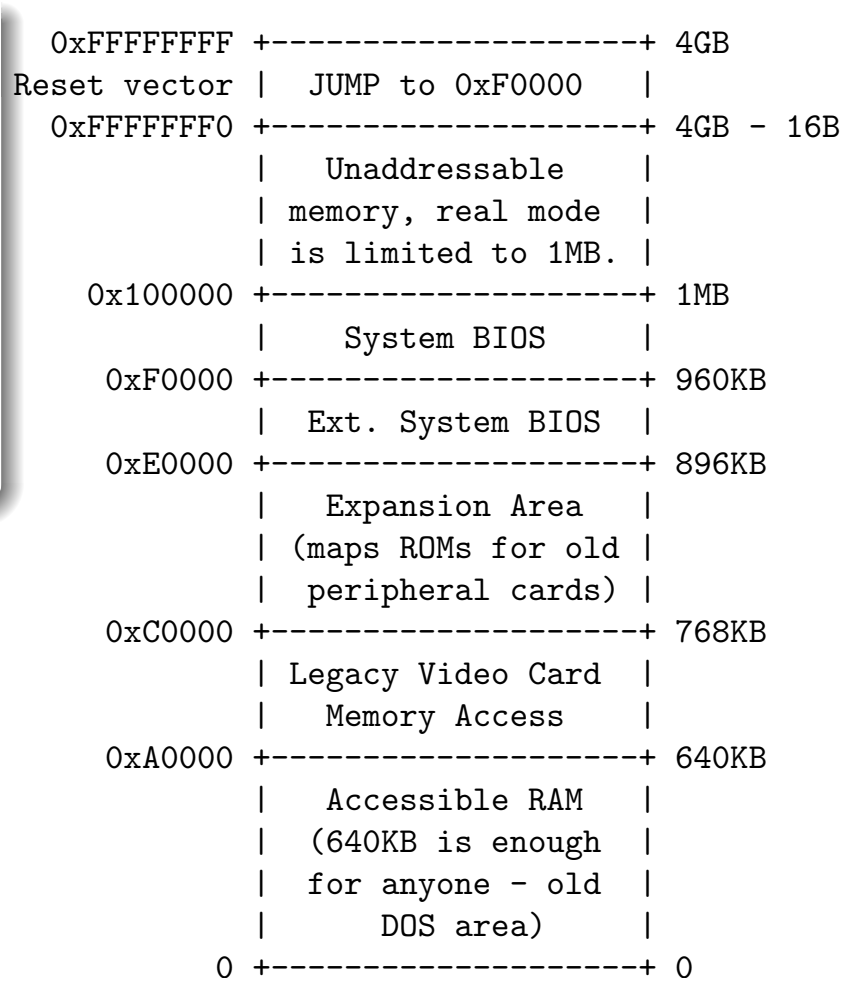
`0x00100000` — The kernel starting point

Reserved page frames

- unavailable to users
- kernel code and data structures
- no dynamic assignment, no swap out

The kernel is loaded starting from the second megabyte (0x00100000) in RAM

- Page frame 0 — BIOS
- $640K \sim 1M$ — the well-know hole
- `/proc/iomem`



While booting

1. The kernel queries the BIOS for available physical address ranges
2. `machine_specific_memory_setup()` — builds the physical addresses map
3. `setup_memory()` — initializes a few variables that describe the kernel's physical memory layout
   - `min_low_pfn`, `max_low_pfn`, `highstart_pfn`, `highend_pfn`, `max_pfn`

BIOS-Provided Physical Addresses Map

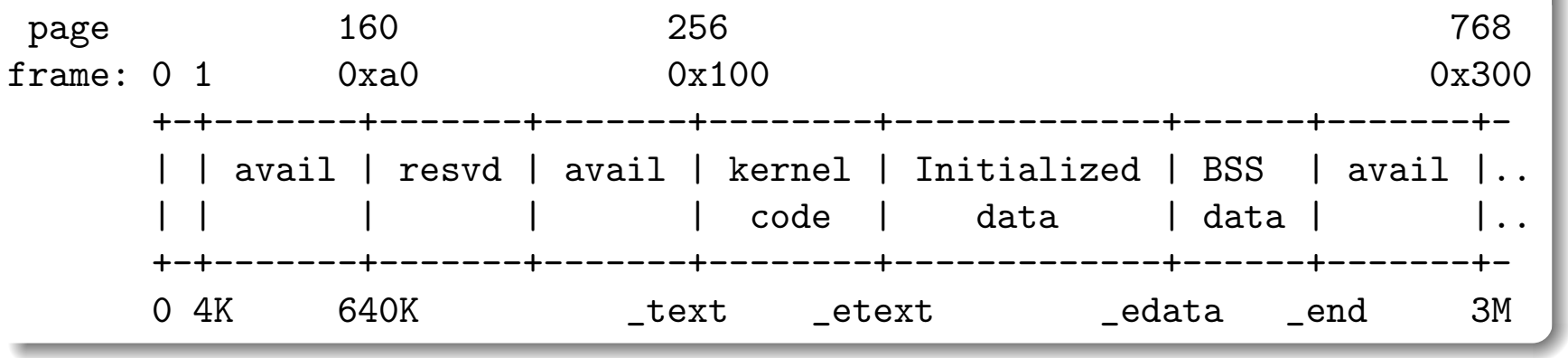
Example — a typical computer with 128MB RAM

Start	End	Type
0x00000000	0x0009ffff (640K)	Usable
0x000f0000 (960K)	0x000fffff (1M-1)	Reserved
0x00100000 (1M)	0x07feffff	Usable
0x07ff0000	0x07ff2fff	ACPI data
0x07ff3000	0x07ffffff (128M)	ACPI NVS
0xfffe0000	0xffffffff	Reserved

Variables describing the physical memory layout

Variable name	Description
`num_physpages`	Page frame number of the highest usable page frame
`totalram_pages`	Total number of usable page frames
`min_low_pfn`	Page frame number of the first usable page frame after the kernel image in RAM
`max_pfn`	Page frame number of the last usable page frame
`max_low_pfn`	Page frame number of the last page frame directly mapped by the kernel (low memory)
`totalhigh_pages`	Total number of page frames not directly mapped by the kernel (high memory)
`highstart_pfn`	Page frame number of the first page frame not directly mapped by the kernel
`highend_pfn`	Page frame number of the last page frame not directly mapped by the kernel

The first 768 page frames (3 MB) in Linux 2.6

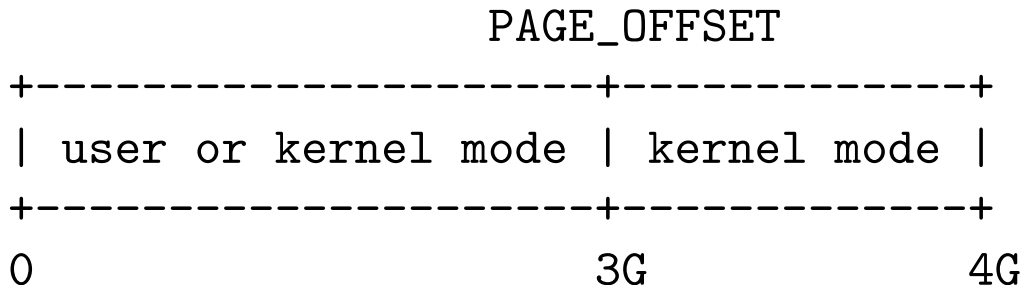


Process Page Tables

0xC0000000 $\Leftrightarrow$ PAGE_OFFSET

include/asm-i386/page.h

```
#define __PAGE_OFFSET           (0xC0000000)
#define PAGE_OFFSET             ((unsigned long)__PAGE_OFFSET)
```



Why?

- easy to switch to kernel mode
- easy physical addressing due to direct mapping

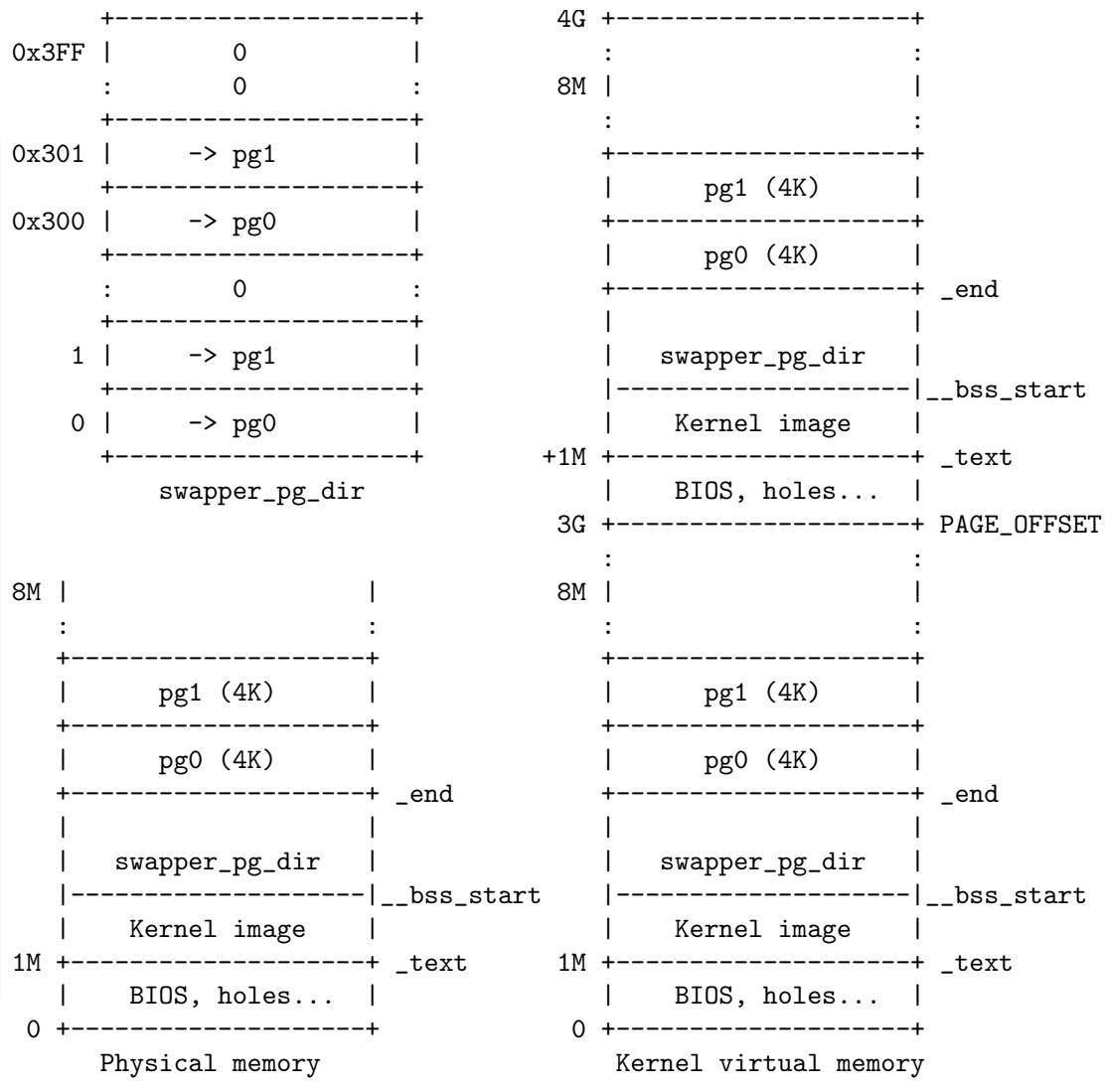
$$Physical = Virtual - \texttt{PAGE\_OFFSET}$$

Kernel Page Tables

— Master Kernel Page Global Directory

Master Kernel PGDir

- has 1K 4-byte entries pointing to 1K page tables
- only 4 entries are used in initialization phase
- after initialization, it's used as a reference model for all processes



In The Beginning, There Is No Paging

Before tuning on paging, the page tables must be ready

Two phases:

1. **Bootstrapping:** sets up page tables for just 8MB so the paging unit can be enabled

   8MB? 2 page tables (pg0, pg1), enough to handle the kernel's code and data segments, and 128 KB for some dynamic data structures (page frame bitmap)

2. **Finalising:** initializes the rest of the page tables

Provisional Page Global Directory

- A provisional PGDir is initialized statically during kernel compilation

```
.section ".bss.page_aligned","w"
ENTRY(swapper_pg_dir)
        .fill 1024,4,0
```

- The provisional PTs are initialized by `startup_32()` in `arch/i386/kernel/head.S`
- `swapper_pg_dir` — A 4KB area for holding provisional PGDir
- provisional PGDir has only 4 useful entries: 0, 1, 0x300, 0x301

What's it for?

Linear		Physical
$0 \sim 8MB$	$\Rightarrow$	$0 \sim 8MB$
$\texttt{PAGE\_OFFSET} \sim (\texttt{PAGE\_OFFSET} + 8MB)$	$\nRightarrow$	

so that the kernel image ($< 8MB$) in physical memory can be addressed in both real mode and protected mode

Provisional Page Table Initialization

arch/i386/kernel/head.S

```
page_pde_offset = (__PAGE_OFFSET >> 20);

        movl $(pg0 - __PAGE_OFFSET), %edi
        movl $(swapper_pg_dir - __PAGE_OFFSET), %edx
        movl $0x007, %eax       # 0x007 = PRESENT+RW+USER
10:
        leal 0x007(%edi), %ecx  # Create PDE entry
        movl %ecx, (%edx)       # Store identity PDE entry
        movl %ecx, page_pde_offset(%edx)  # Store kernel PDE entry
        addl $4, %edx
        movl $1024, %ecx
11:
stosl   # movl %eax, (%edi)
        # addl £4, %edi
        addl $0x1000, %eax
        loop 11b
        # End condition: we must map up to and including INIT_MAP_BEYOND_END
        # bytes beyond the end of our own page tables; the +0x007 is the
        # attribute bits
        leal (INIT_MAP_BEYOND_END + 0x007)(%edi), %ebp
        cmpl %ebp, %eax
        jb 10b
        movl %edi, (init_pg_tables_end - __PAGE_OFFSET)
```

Equivalent pseudo C code

```
/*
 * Provisional PGDir and page tables setup
 *
 * for mapping two linear address ranges to the same physical address range
 *
 *  + Linear address ranges:
 *              -   User mode: i × 4M ∼ (i + 1) × 4M − 1
 *              - Kernel mode: 3G + i × 4M ∼ 3G + (i + 1) × 4M − 1
 *  + Physical address range: i × 4M ∼ (i + 1) × 4M − 1
 */
typedef unsigned int PTE;
PTE *pg = pg0;      /* physical address of pg0 */
PTE pte = 0x007;    /* 0x007 = PRESENT+RW+USER */
for(i=0;;i++){
  swapper_pg_dir[i] = pg + 0x007;          /* store identity PDE entry */
  swapper_pg_dir[i+page_pde_offset] = pg + 0x007; /* kernel PDE entry */
  for(j=0;j<1024;j++){                    /* populating one page table */
    pg[i*1024 + j] = pte;                 /* fill up one page table entry */
    pte += 0x1000;                        /* next 4k */
  }
  if(pte >= ((char*)pg + i*1024 + j)*4 + 0x007 + INIT_MAP_BEYOND_END)
    {
      init_pg_tables_end = pg + i*0x1000 + j;
      break;
    }
  }
```

Enable paging

`startup_32()` in `arch/i386/kernel/head.S`

```
# Enable paging
movl $swapper_pg_dir - __PAGE_OFFSET, %eax
movl %eax, %cr3          # set the page table pointer..
movl %cr0, %eax
 orl $0x80000000, %eax
movl %eax, %cr0          # ..and set paging (PG) bit
```

Final Kernel Page Table Setup

- `master kernel PGDir` is still in `swapper_pg_dir`
- initialized by `paging_init()`

Situations

1. `RAM size` $< 896M$
   - every RAM cell is mapped
2. $896M <$ `RAM size` $< 4G$
   - 896M are mapped
3. `RAM size` $> 4G$
   - PAE enabled

When RAM size is less than 896 MB

`paging_init()` without PAE

```
1  void __init paging_init(void)
2  {
3  #ifdef CONFIG_X86_PAE
4    /* ... */
5  #endif
6
7    pagetable_init();
8    load_cr3(swapper_pg_dir);
9
10 #ifdef CONFIG_X86_PAE
11   /* ... */
12 #endif
13
14   __flush_tlb_all();
15   kmap_init();
16   zone_sizes_init();
17 }
```

2 level paging: PUD and PMD are folded

```
+---------+---------+---------+---------+------------+
| Global  | Upper   | Middle  | Page    | Offset     |
| dir   10| dir    0| dir    0| tbl   10|          12|
+---------+---------+---------+---------+------------+
```

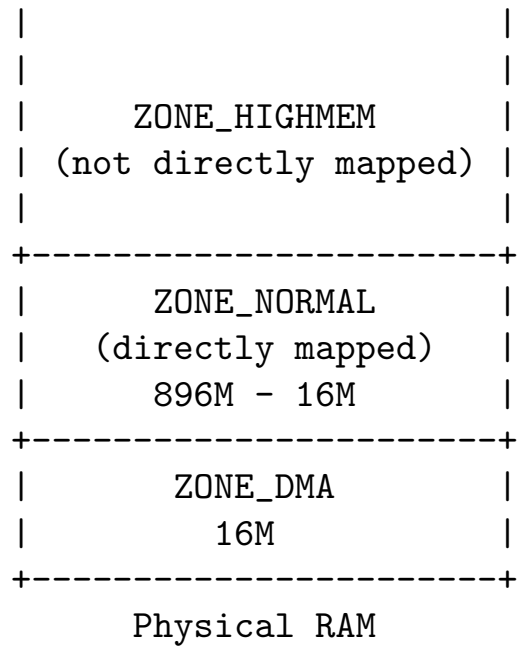
`pagetable_init()` — re-initializes the `PGDir` at `swapper_pg_dir`

Equivalent code:

```
1  pgd = swapper_pg_dir + pgd_index(PAGE_OFFSET); /* 768 */
2  phys_addr = 0x00000000;
3  while (phys_addr < (max_low_pfn * PAGE_SIZE))
4  {
5    pmd = one_md_table_init(pgd); /* returns pgd itself */
6    set_pmd(pmd, __pmd(phys_addr | pgprot_val(__pgprot(0x1e3))));
7    /* 0x1e3 == Present, Accessed, Dirty, Read/Write, Page Size, Global */
8    phys_addr += PTRS_PER_PTE * PAGE_SIZE; /* 0x400000, 4M */
9    ++pgd;
10 }
```

When RAM Size Is Between 896MB ∼ 4096MB

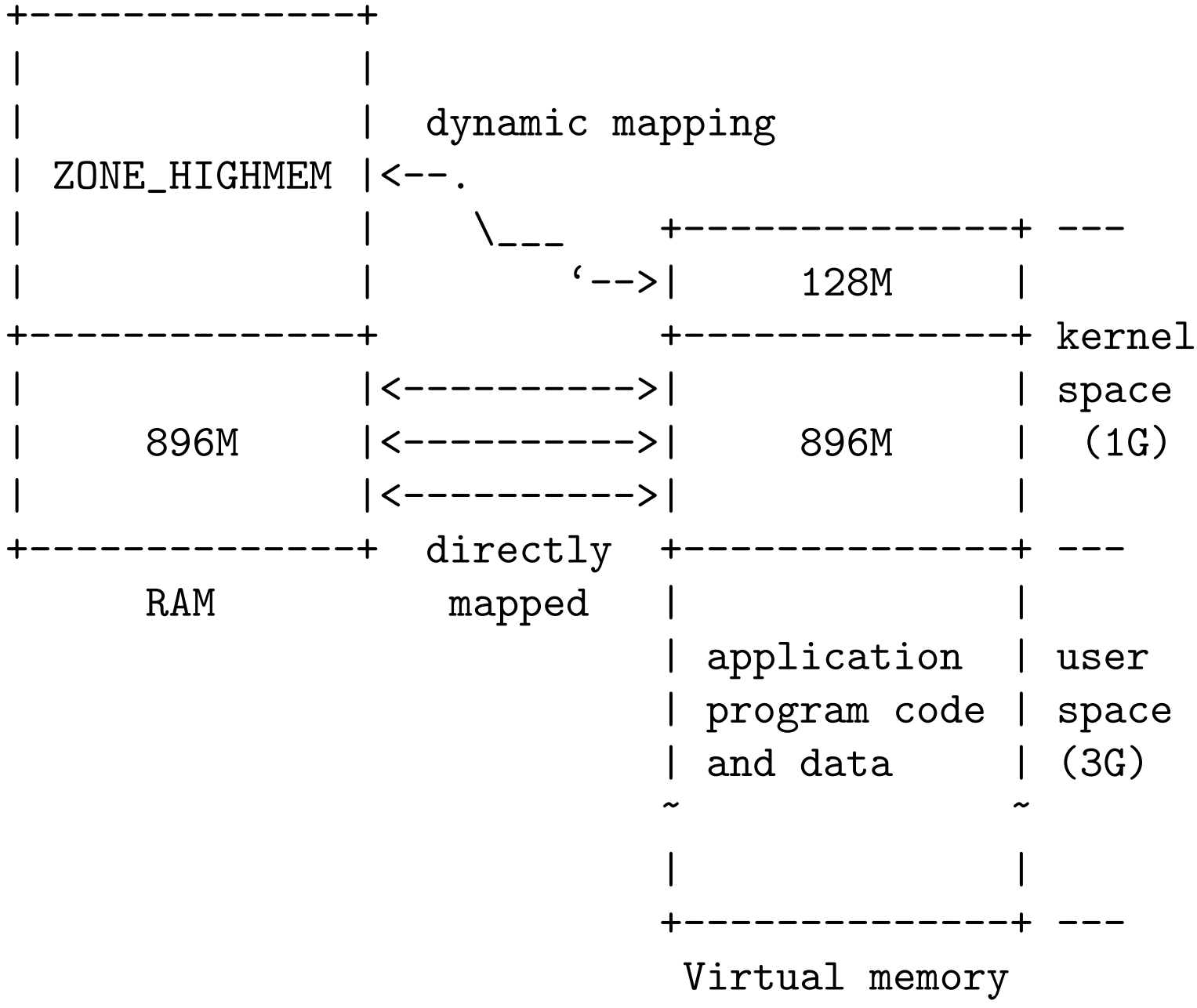
Physical memory zones:



Direct mapping for ZONE_NORMAL:

```
#define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

High Memory


ZONE_HIGHMEM
dynamic mapping
128M
kernel space (1G)
896M
896M
directly mapped
RAM
application program code and data
user space (3G)
Virtual memory

When RAM Size Is More Than 4096MB (PAE)

A 3-level paging model is used

```
        3-level paging for 4K-pages
+--+---------+---------+------------+
|PD|  Page   |  Page   |   Offset   |
|PT|  DIR    |  Table  |            |
+--+---------+---------+------------+
 2      9         9           12
```

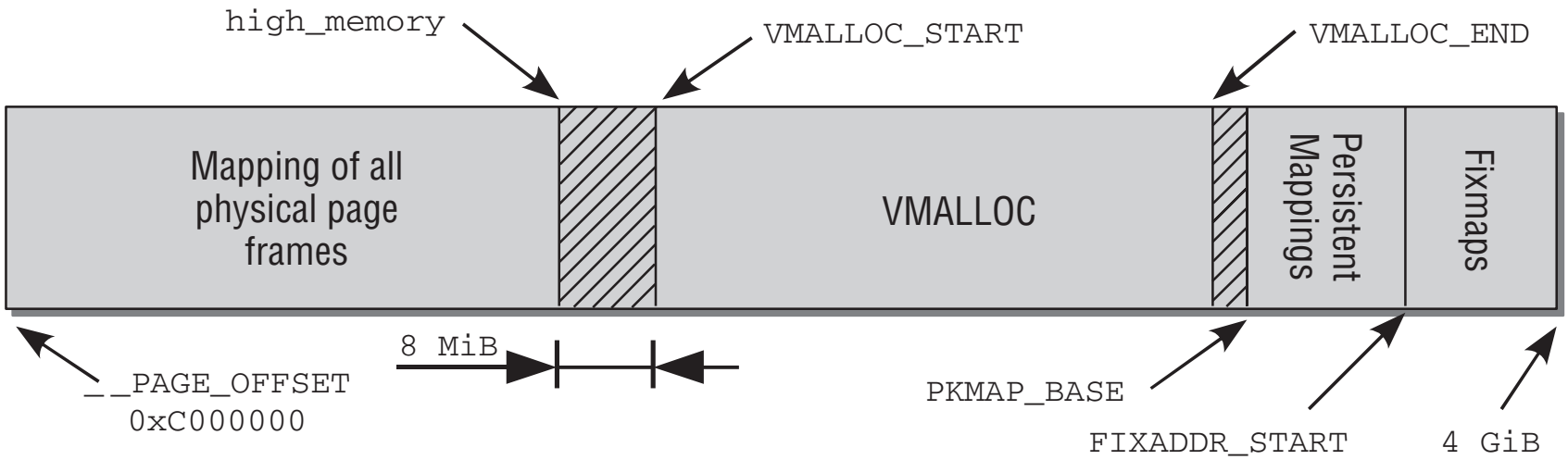
PGDir	PUD	PMD	PT	OFFSET
2	0	9	9	12

The PGDir is initialized by a cycle equivalent to the following:

```
pgd_idx = pgd_index(PAGE_OFFSET); /* 3 */
for (i=0; i<pgd_idx; i++)
  set_pgd(swapper_pg_dir + i, __pgd(__pa(empty_zero_page) + 0x001));
  /* 0x001 == Present */
pgd = swapper_pg_dir + pgd_idx;
phys_addr = 0x00000000;
for (; i<PTRS_PER_PGD; ++i, ++pgd) {
  pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
  set_pgd(pgd, __pgd(__pa(pmd) | 0x001)); /* 0x001 == Present */
  if (phys_addr < max_low_pfn * PAGE_SIZE)
    for (j=0; j < PTRS_PER_PMD /* 512 */
      && phys_addr < max_low_pfn*PAGE_SIZE; ++j) {
      set_pmd(pmd, __pmd(phys_addr | pgprot_val(__pgprot(0x1e3))));
      /* 0x1e3 == Present, Accessed, Dirty, Read/Write,
         Page Size, Global */
      phys_addr += PTRS_PER_PTE * PAGE_SIZE; /* 0x200000 */
    }
 }
swapper_pg_dir[0] = swapper_pg_dir[pgd_idx];
```

Division Of The Kernel Address Space

— On IA-32 Systems



- Virtually contiguous memory areas that are *not* contiguous in physical memory can be reserved in the vmalloc area.
- *Persistent mappings* are used for persistent kernel mapping of highmem page frames.
- *Fixmaps* are virtual address space entries associated with a fixed but freely selectable page in physical address space.