Memory Addressing

Wang Xiaolin
wx672ster+os@gmail.com

May 19, 2014

Contents

Textbook:

- *Memory Addressing*, [BC05, Chapter 2]
- *Memory management in Linux*, [NGC02]
- *Understanding The Linux Virtual Memory Manager*, [Gor04]
- *Memory Management*, [Mau08, Chapter 3]
- *Virtual Process Memory*, [Mau08, Chapter 4]
- *Memory Management*, [Lov10, Chapter 12]
- *The Process Address Space*, [Lov10, Chapter 15]

References

[BC05] D.P. Bovet and M. Cesatí. *Understanding The Linux Kernel*. 3rd ed. O'Reilly, 2005.

[Gor04] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.

[Int86] Intel. *INTEL 80386 Programmer's Reference Manual*. 1986.

[Lov10] R. Love. *Linux Kernel Development*. Developer's Library. Addison-Wesley, 2010.

[Mau08] W. Mauerer. *Professional Linux Kernel Architecture*. John Wiley & Sons, 2008.

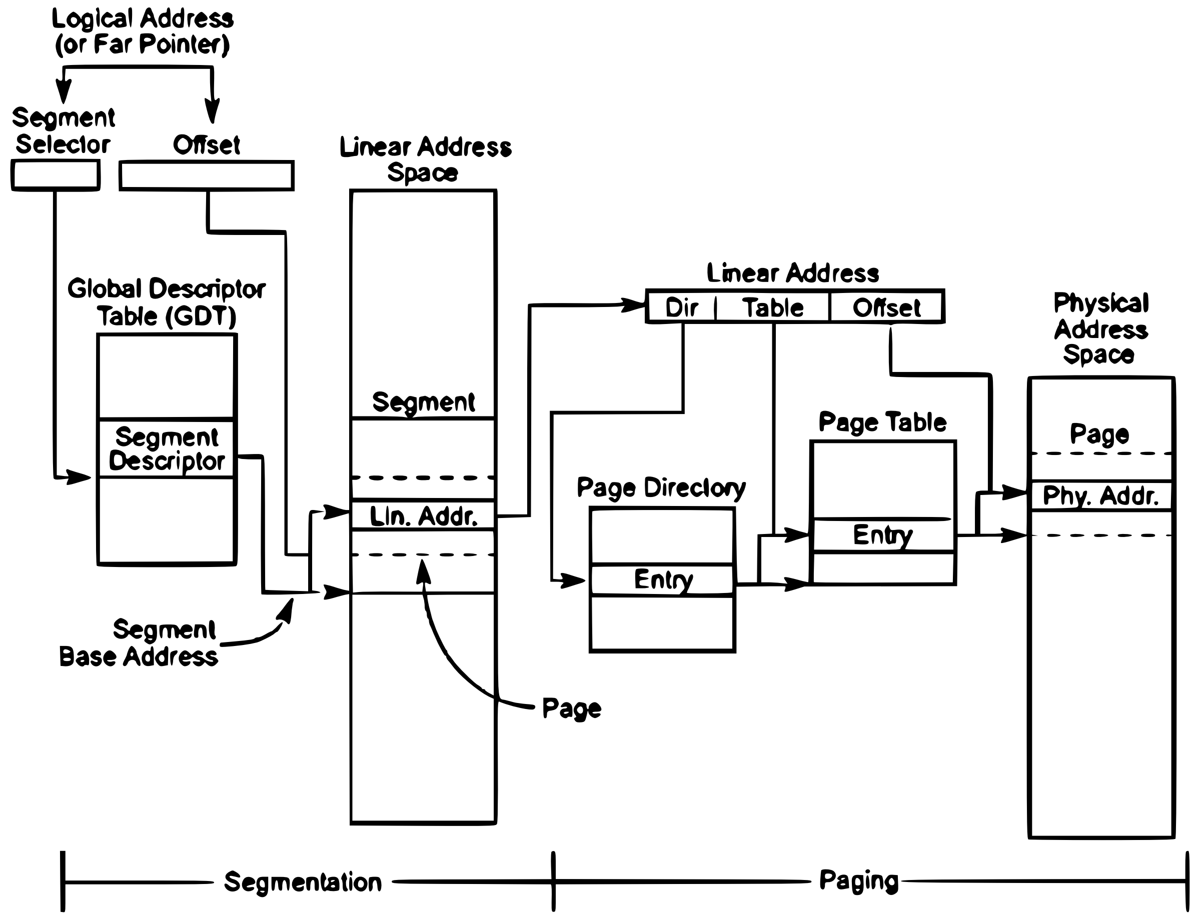
[NGC02] Abhishek Nayani, Mel Gorman, and Rodrigo S. de Castro. *Memory Management in Linux: Desktop Companion to the Linux Source Code*. Free book, 2002.

[王爽 03] 王爽. 汇编语言. 清华大学出版社, 2003.

1 Memory Addresses

Three Kinds Of Addresses

```
                 |<--------------- MMU ---------------->|
                  +---------------+           +--------+            +----------+
+-----+  Logical  | Segmentation  |  Linear   | Paging |  Physical  | Physical |
| CPU |---------->|     unit      |---------->|  unit  |----------->|  memory  |
+-----+  address  +---------------+  address  +--------+  address   +----------+
```



Picture source: http://ilinuxkernel.com/wp-content/uploads/2011/09/091011_1614_Linux1.png

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*.

2 Segmentation in Hardware

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

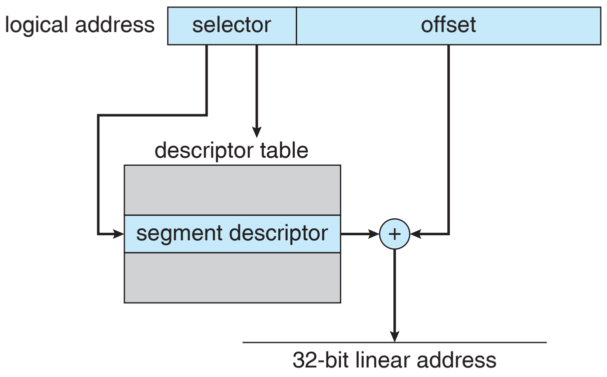
$$segment\ number \times 2^4 + offset$$

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

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

- *16-bit CPU*, ([王爽 03, Sec 2.4]).

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

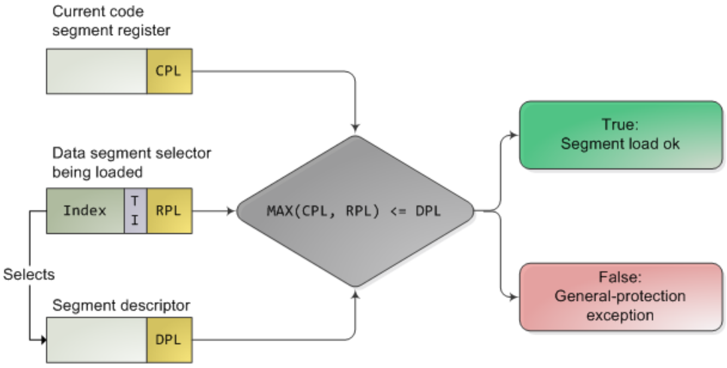
ss stack segment register

ds data segment register

es/fs/gs general purpose registers, may refer to arbitrary data segments

More about privilege levels:

- CPU Rings, Privilege, and Proctection
- *Segment-Level Protection*, [Int86, Sec 6.3]


Current code segment register
CPL
Data segment selector being loaded
Index
TI
RPL
MAX(CPL, RPL) <= DPL
Selects
Segment descriptor
DPL
True: Segment load ok
False: General-protection exception

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

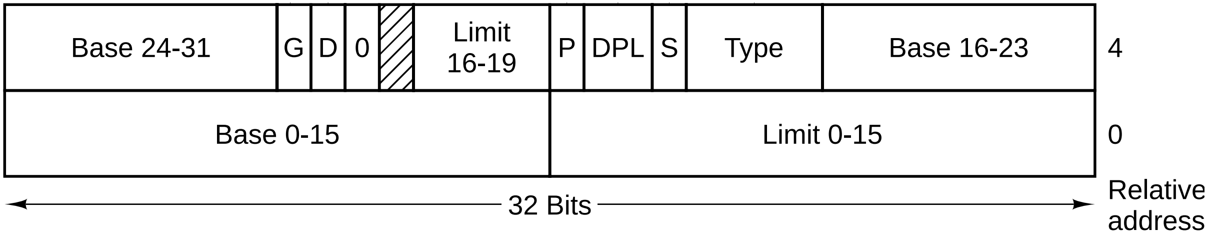
Analogy

Process	⟺	Process Descriptor(PCB)
File	⟺	Inode
Segment	⟺	Segment Descriptor

More info:

- Memory Tanslation And Segmentation
- `http://www.osdever.net/bkerndev/Docs/gdt.htm`
- Sec 4 of *JamesM's kernel development tutorials*, The GDT and IDT

Example: A LDT entry for code segment



Base: Where the segment starts

Limit: 20 bit, ⇒ 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

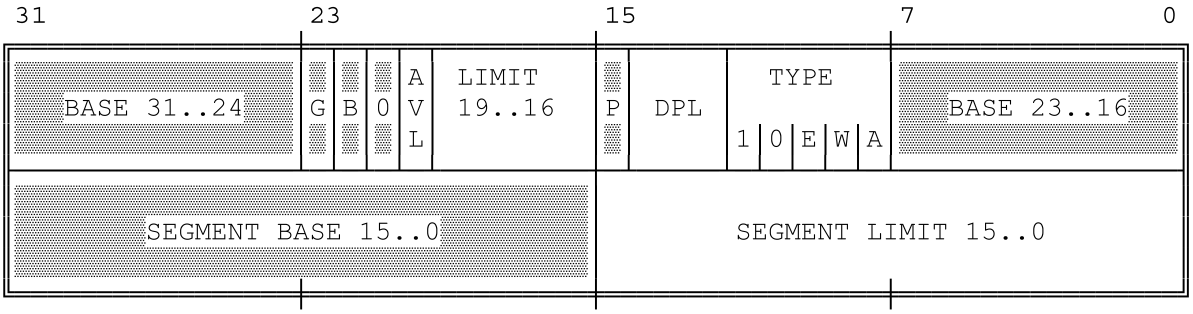
0 - not in memory

1 - in memory

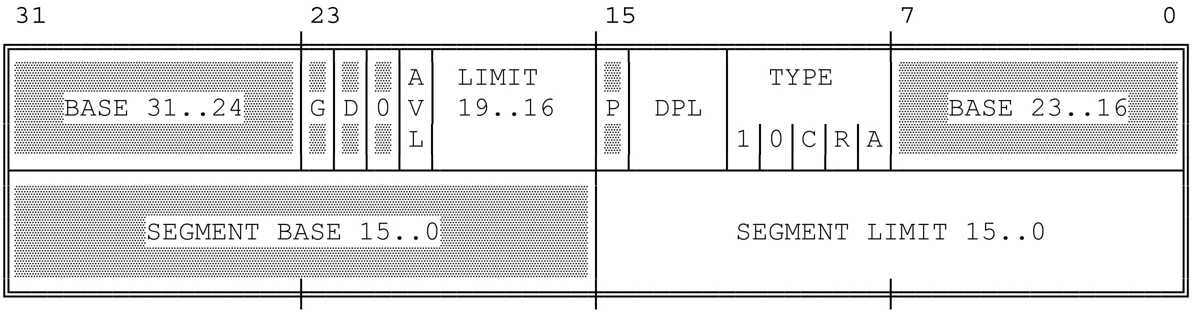
AVL: ignored by Linux

- *Segment-Level Protection*, [Int86, Sec 6.3, p108]

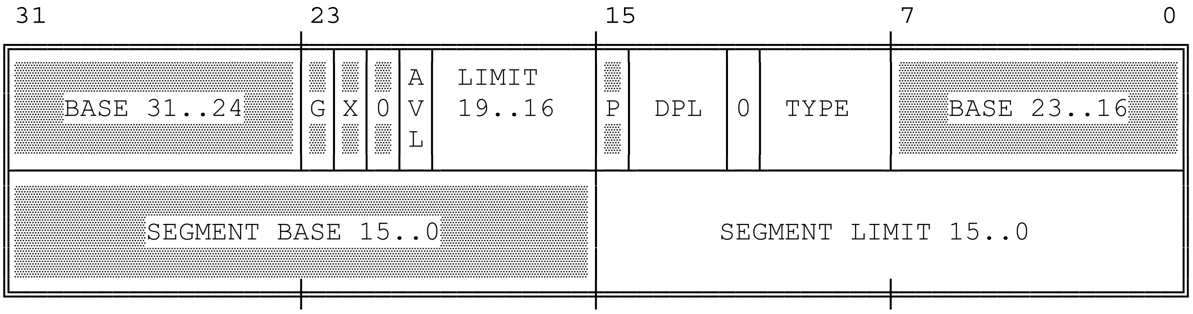
DATA SEGMENT DESCRIPTOR



EXECUTABLE SEGMENT DESCRIPTOR



SYSTEM SEGMENT DESCRIPTOR



A	- ACCESSED
AVL	- AVAILABLE FOR PROGRAMMERS USE
B	- BIG
C	- CONFORMING
D	- DEFAULT
DPL	- DESCRIPTOR PRIVILEGE LEVEL
E	- EXPAND-DOWN
G	- GRANULARITY
P	- SEGMENT PRESENT
R	- READABLE
W	- WRITABLE

Fast Access to Segment Descriptors

a non-programmable cache register for each segment register

```
      DESCRIPTOR TABLE              SEGMENT
      +---------------+             +-------+
      |      ...      |    ,------->|       |<----.
      +---------------+    |        |       |     |
 .--->|   Segment     |____/        |       |     |
|     |  Descriptor   |             +-------+     |
|     +---------------+                           |
|     |      ...      |                           |
|     +---------------+       Nonprogrammable     |
|                                Register         |
|    Segment Registor      +--------------------+_/
\__+-------------------+   | Segment Descriptor |
   | Segment Selector |    +--------------------+
   +-------------------+
```

Translating a logical address

```
              GDT or LDT
            +------------+
            |    ...     |         +---------+
            +------------+         | Linear  |
   .------->| Descriptor |---(+)-->| Address |
  |         +------------+    ^    +---------+
  |         |    ...     |    |
  |    .--->+------------+    |
  |    |base                  |
  |    |                      |
  |    |   +--------------+   |
 (+)<-----| GDTR or LDTR |    |
  ^       +--------------+    |
  |         ^                 |
  |         |                 |
 (x8)       |                 |
  ^         |                 |
  |         |                 |
+----------+-+--+-----------------------------+
|  Index   |g|p |            offset            |
+----------+-+--+-----------------------------+
             Logical Address
```

1. $Index \times 8 + table\ base$
2. $Descriptor\ base + offset$

3 Segmentation in Linux

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

- (*Task State Segment*, [BC05, Sec 3.3.2]) Although Linux doesn't use hardware context switches, it is nonetheless forced to set up a TSS for each distinct CPU in the system. This is done for two main reasons:
  - When an 80x86 CPU switches from User Mode to Kernel Mode, it fetches the address of the Kernel Mode stack from the TSS (see the sections "Hardware Handling of Interrupts and Exceptions" in Chapter 4 and "Issuing a System Call via the sysenter Instruction" in Chapter 10). (Wikipedia: Inner-level stack pointers)
  - When a User Mode process attempts to access an I/O port by means of an in or out instruction, the CPU may need to access an I/O Permission Bitmap stored in the TSS to verify whether the process is allowed to address the port. (Wikipedia: I/O port permissions)
- Wikipedia: Task State Segment
- Wikipedia: Local Descriptor Table
- *The Linux LDTs*, [BC05, Sec 2.3.2]

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

- Values in segment registers:

```
|          Index          |G:RPL|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 1 1 1 0|0|1 1| __USER_CS   (14 << 3 + 3)
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 1 1 1 1|0|1 1| __USER_DS   (15 << 3 + 3)
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 1 1 0 0|0|0 0| __KERNEL_CS (12 << 3 + 0)
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 1 1 0 1|0|0 0| __KERNEL_DS (13 << 3 + 0)
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- *Segmentation in Linux*, [BC05, Sec 2.3]

  The corresponding Segment Selectors are defined by the macros `__USER_CS`, `__USER_DS`, `__KERNEL_CS`, and `__KERNEL_DS`, respectively. To address the kernel code segment, for instance, the kernel just loads the value yielded by the `__KERNEL_CS` macro into the `cs` segmentation register.

  Notice that the linear addresses associated with such segments all start at `0` and reach the addressing limit of $2^{32} - 1$. This means that all processes, either in User Mode or in Kernel Mode, may use the same logical addresses.

  Another important consequence of having all segments start at `0x00000000` is that in Linux, logical addresses coincide with linear addresses; that is, the value of the Offset field of a logical address always coincides with the value of the corresponding linear address.

  As stated earlier, the Current Privilege Level of the CPU indicates whether the processor is in User or Kernel Mode and is specified by the RPL field of the Segment Selector stored in the `cs` register. *Whenever the CPL is changed, some segmentation registers must be correspondingly updated.* For instance, when the CPL is equal to 3 (User Mode), the `ds` register must contain the Segment Selector of the user data segment, but when the CPL is equal to 0, the `ds` register must contain the Segment Selector of the kernel data segment.

  A similar situation occurs for the `ss` register. It must refer to a User Mode stack inside the user data segment when the CPL is 3, and it must refer to a Kernel Mode stack inside the kernel data segment when the CPL is 0. *When switching from User Mode to Kernel Mode, Linux always makes sure that the `ss` register contains the Segment Selector of the kernel data segment.*

  When saving a pointer to an instruction or to a data structure, the kernel does not need to store the Segment Selector component of the logical address, because the `ss` register contains the current Segment Selector. As an example, when the kernel invokes a function, it executes a `call` assembly language instruction specifying just the Offset component of its logical address; the Segment Selector is implicitly selected as the one referred to by the `cs` register. Because there is just one segment of type ”executable in Kernel Mode,” namely the code segment identified by `__KERNEL_CS`, it is sufficient to load `__KERNEL_CS` into `cs` whenever the CPU switches to Kernel Mode. The same argument goes for pointers to kernel data structures (implicitly using the `ds` register), as well as for pointers to user data structures (the kernel explicitly uses the `es` register).

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

```
  |31           24|23   20|19   16|15            8|7             0|
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
7 |  BASE 31..24  |1|1|0|0| LIMIT |1|DPL|  TYPE  |   BASE 23..16   | 4
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
3 |      SEGMENT BASE 15..0       |      SEGMENT LIMIT 15..0        | 0
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 */
```

- $0x60 = 12 \ll 3 + 0$
- $0x68 = 13 \ll 3 + 0$
- $0x73 = 14 \ll 3 + 3$
- $0x7b = 15 \ll 3 + 3$

From the comments of Memory Tanslation And Segmentation:

Q: I went to where the `gdt_page` is instantiated (line 24, `common.c`, 2.6.25)

It has the following code:

```
[GDT_ENTRY_DEFAULT_USER_CS] =    0x0000ffff, 0x00cffa00
```

Do you know what that means?

A: This line is building the 8-byte segment descriptor for the user code segment. To really follow it, there are 3 things you must bear in mind:

1. The x86 is little endian, meaning that for multi-byte data types (say, 32-bit or 64-bit integers), the significance of bytes grows with memory address. If you declare a 32-bit integer as `0xdeadbeef`, then it would be laid out in memory like this (in hex, assuming memory addresses are growing to the right):

   ```
   ef be ad de
   lower => higher
   ```

2. In array declarations, or in this case a struct declaration, earlier elements go into lower memory addresses.
3. The convention for Intel diagrams is to draw things with HIGHER memory addresses on the LEFT and on TOP. This is a bit counter intuitive, but I followed it to be consistent with Intel docs.

When you put this all together, the declaration above will translate into the following bytes in memory, using Intel' s 'reversed' notation:

```
(higher)
00 cf fa 00
00 00 ff ff
(lower)
```

If you compare these values against the segment descriptor diagram above, you’ ll see that: the ‘base’ fields are all zero, so the segment starts at 0×00000000, the limit is 0xfffff so the limit covers 4GB, byte 47-40 is 11111010, so the DPL is 3 for ring 3.

If you look into the Intel docs, they describe the fields I left grayed out. Hope this helps!

```
cpu_gdt_descr         cpu_gdt_table        GDT     Selector
+-----------+   _        +-----+     ---+--------+
|           |->|_|       |     |    / 0 |  null  | 0x0
|           |            +-----+   /    +--------+
+-----------+            |     |  /     |  ...   |
|   size    |   +----+   +-----+-'      +--------+
|  address  |-->|gdtr|-->| GDT |      12| kernel | __KERNEL_CS
+-----------+   +----+   +-----+-.      |  code  | 0x60
|           |   _        :     :  \     +--------+
|           |->|_|       +-----+   \  13| kernel | __KERNEL_DS
+-----------+                       |   |  data  | 0x68
:           :                       |   +--------+
:           :                       | 14|  user  | __USER_CS
+-----------+                       |   |  code  | 0x73
                                    |   +--------+
                                    | 15|  user  | __USER_DS
                                    |   |  data  | 0x7b
                                    |   +--------+
                                    |   |  ...   |
                                    `---+--------+
```

cpu_gdt_table: is an array of all GDTs

cpu_gdt_descr: store the addresses and sizes of the GDTs

4 Paging in Hardware

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

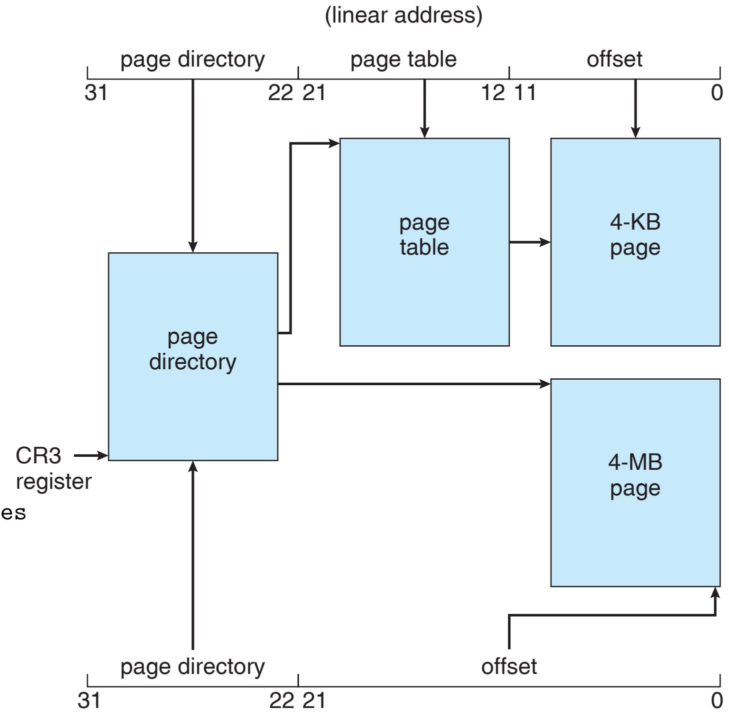
- is a data structure
- maps linear to physical addresses
- stored in main memory

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
```



- The CR3 register points to the top level page table for the current process.

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

```
 31                                  12 11                      0
+--------------------------------------+-------+---+-+-+---+-+-+-+
|                                      |       |   | | |   |U|R| |
|      PAGE FRAME ADDRESS 31..12       | AVAIL |0 0|D|A|0 0|/|/|P|
|                                      |       |   | | |   |S|W| |
+--------------------------------------+-------+---+-+-+---+-+-+-+
        P      - PRESENT
        R/W    - READ/WRITE
        U/S    - USER/SUPERVISOR
        A      - ACCESSED
        D      - DIRTY
        AVAIL  - AVAILABLE FOR SYSTEMS PROGRAMMER USE

        NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

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

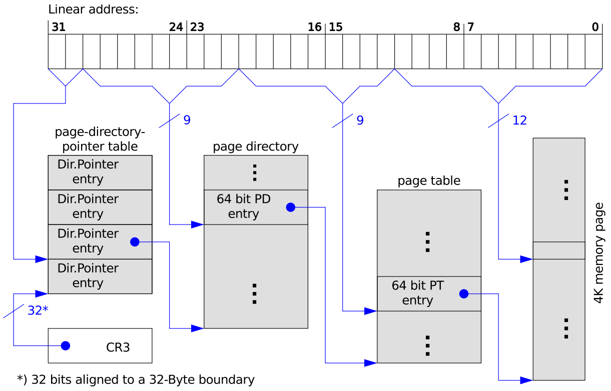
```
    3-level paging for 4K-pages
+--+---------+---------+------------+
|PD|  Page   |  Page   |   Offset   |
|PT|  DIR    |  Table  |            |
+--+---------+---------+------------+
 2      9         9          12
```

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

`64-bit entry` $\times 4$

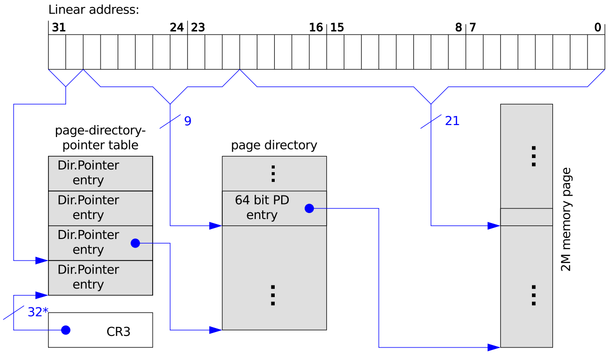
```
    2-level paging for 2M-pages
+--+---------+----------------------+
|PD|  Page   |       Offset         |
|PT|  DIR    |                      |
+--+---------+----------------------+
 2      9              21
```

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)

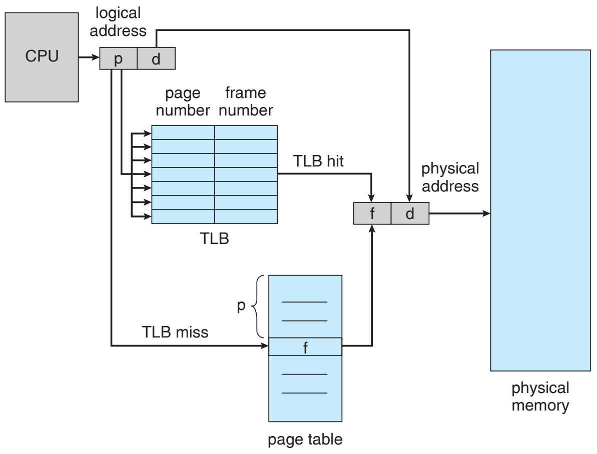
The linear address are still 32 bits

- 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

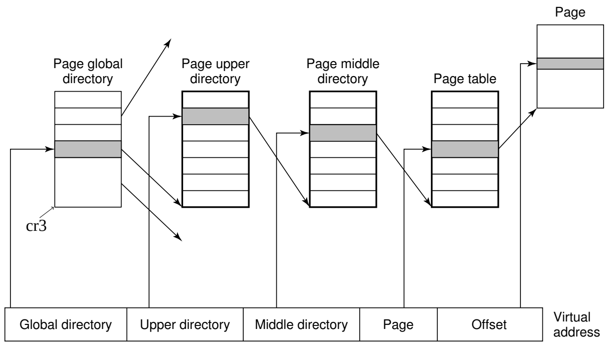
- Only a small fraction of the PTEs are heavily read; the rest are barely used at all



5 Paging in Linux

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	$9 + 9 + 9 + 9 + 12$

ℹ

(*Paging In Linux*, [BC05, Sec 2.5]) For 32-bit architectures with no PAE, two paging levels are sufficient. Linux essentially eliminates the PUD and the PMD fields by saying that they contain zero bits. However, the positions of the PUD and the PMD in the sequence of pointers are kept so that the same code can work on 32-bit and 64-bit architectures.

Where are *the sequence of pointers* kept? The kernel keeps a position for the PUD and the PMD by setting the number of entries in them to 1 and mapping these two entries into the proper entry of the Page Global Directory.

How can Linux use 4-level model, while Intel specifies 2-level paging? (32-bit no PAE) Quoted from `http://stackoverflow.com/questions/6627833/process-page-tables`:

> What I was trying to say was that the paging model used by the OS and the model used by the hardware are often distinct concepts. Linux uses a uniform paging model internally but this is layered on top of the hardware's paging model and requires architecture specific hacks to get it to work. It is the hardware's model that determines how address translation actually occurs (since it is the MMU that does this). *Linux simply adds a layer of indirection on top. Deep in its bowels it still uses the 10:10:12 model*. — *Abhay Buch Jul 8 '11 at 22:58*

More about paging Quoted from `http://linux-mm.org/VirtualMemory`:

> after bootup, all memory is accessed through the MMU which is paging enabled. So everything before and after `PAGE_OFFSET` is paged. not everything can be demand paged, though ... (since kernel data structures are resident). pages below `PAGE_OFFSET` belong to user space, and can be demand paged. addresses above `PAGE_OFFSET` are kernel memory. there is a linear mapping for the first 900 MB of kernel memory, where physical address 0 - 896 MB is mapped into `PAGE_OFFSET` - `PAGE_OFFSET`+896MB. so there are 896M/4K physical frames addressable from `PAGE_OFFSET`->`PAGE_OFFSET`+896MB.
>
> memory above `PAGE_OFFSET` is kernel virtual memory. part of it is a direct map of the first part of physical memory. *but that same physical memory could also get virtual mappings from elsewhere*, eg. userspace or vmalloc. also, userspace and vmalloc can map physical memory from outside the 896MB of direct mapped memory (as well as inside it).
>
> most kernel memory allocation needs to come from that 896 MB, indeed, though page tables are the big exception ;) which means they're resident in memory all the time. kernel data structures are always resident.
>
> process page tables could be either inside the low 896MB, or in highmem (or some page tables in both - more likely).
>
> physical memory is, by definition, not pageable. the contents of those pages might be pageable though. so you could have a page P at physical address 400MB. a process (eg. mozilla) is using that page at virtual address 120MB somewhere in its heap. the contents of the physical page can be paged out, at which point mozilla's heap page at 120MB is paged out. but the kernel mapping (at `PAGE_OFFSET` + 400MB) still maps the same page P just with different contents ;)

5.1 The Linear Address Fields

The Linear Address Fields

```
|<-PGDIR_MASK->|<------------------PGDIR_SIZE----------------->|
|<--------PUD_MASK------->|<-------------PUD_SIZE------------->|
|<-------------PMD_MASK------------->|<-------PMD_SIZE-------->|
|<-----------------PAGE_MASK------------------->|<-PAGE_SIZE-->|
+--------------+----------+----------+----------+--------------+
| Global       | Upper    | Middle   | Page     | Offset       |
| DIR         ?| DIR     ?| DIR     ?| Table   ?|            12|
+--------------+----------+----------+----------+--------------+
                                                |<-PAGE_SHIFT->|
                                     |<---PMD_SHIFT----------->|
                          |<-----PUD_SHIFT-------------------->|
               |<-----PGDIR_SHIFT----------------------------->|
```

`*_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`

ℹ

`PAGE_SHIFT`: Specifies the size of the area that a *page table entry* can cover

`PMD_SHIFT`: Specifies the size of the area that a *PMD entry* can cover

- When PAE is disabled, `PMD_SHIFT` yields the value 22 (12 from Offset plus 10 from Table)
- when PAE is enabled, `PMD_SHIFT` yields the value 21 (12 from Offset plus 9 from Table)
- `LARGE_PAGE_SIZE = PMD_SIZE`
  - 2^{22}, without PAE
  - 2^{21}, with PAE

`PUD_SHIFT`: Specifies the size of the area that a *PUD entry* can cover

- On the 80x86 processors, `PUD_SHIFT` is always equal to `PMD_SHIFT`
  - because both PUD and PMD are 0-bit long
- `PUD_SIZE` is equal to 4MB or 2MB.

`PGDIR_SHIFT`: Specifies the size of the area that a *PGDIR entry* can cover

- When PAE is disabled, `PGDIR_SHIFT = PUD_SHIFT = PMD_SHIFT = 22`
- when PAE is enabled, `PGDIR_SHIFT` $= 9_{PMD} + 9_{PT} + 12_{Offset} = 30$

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

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

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

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

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

```
+---------+---------+---------+---------+------------+
| Global  | Upper   | Middle  | Page    | Offset     |
| DIR    9| DIR    9| DIR    9| Table  9|          12|
+---------+---------+---------+---------+------------+
```

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
```

5.2 Page Table Handling

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
```

Why structs? ([Mau08, p158]) `structs` are used instead of elementary types to ensure that the contents of page table elements are handled only by the associated helper functions and never directly.

(*Describing a Page Table Entry*, [Gor04, Sec 3.2]) Even though these are often just unsigned integers, they are defined as structs for two reasons. The first is for type protection so that they will not be used inappropriately. The second is for features like PAE...

- For 32-bit systems,
  - long int: 4 bytes
  - long long int: 8 bytes
- `pgprot_t` holds the protection flags associated with a single entry.
  - Present/RW/User/Accessed/Dirty...
- `__pmd(x)`, type casting
- `pmd_val(x)`, reverse casting
- `HPAGE_SHIFT`, huge page shift.
  - 22 - without PAE, page size is $2^{22} = 4M$
  - 21 - with PAE, page size is $2^{21} = 2M$

Page Table Handling

— *Read or modify page table entries*

Macros and functions

pte_none
pte_present
pte_exec()
mk_pte_huge()
pte_mkwrite()
pte_mkdirty()
mk_pte(p,prot)

pte_clear
pte_user()
pte_dirty()
pte_wrprotect()
pte_mkread()
pte_mkold()
pte_index(addr)

set_pte
pte_read()
pte_young()
pte_rdprotect()
pte_mkexec()
pte_mkyoung()
pte_page(x)

pte_same(a,b)
pte_write()
pte_file()
pte_exprotect()
pte_mkclean()
pte_modify(p,v)
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;
```

pgd_offset(mm, addr) Receives as parameters the address of a memory descriptor mm (*Process Address Space*, [BC05, Chapter 9]) and a linear address addr. The macro yields the linear address of the entry in a Page Global Directory that corresponds to the address addr; the Page Global Directory is found through a pointer within the memory descriptor.

Memory descriptor (*The Memory Descriptor*, [BC05, Sec 9.2]) All information related to the process address space is included in an object called the memory descriptor of type mm_struct. This object is referenced by the mm field of the process descriptor.

Line 312 in pgtable.h:

```
/*
 * pgd_offset() returns a (pgd_t *)
 * pgd_index() is used get the offset into the pgd page's array of pgd_t's;
 */
#define pgd_offset(mm, address) ((mm)->pgd+pgd_index(address))
```

Line 305 in pgtable.h:

```
/*
 * the pgd page can be thought of an array like this: pgd_t[PTRS_PER_PGD]
 *
 * this macro returns the index of the entry in the pgd page which would
 * control the given virtual address
 */
#define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD-1))
```

$$\mathtt{PTRS\_PER\_PGD} = \begin{cases} 1024 & \mathtt{i386, noPAE} \\ 4 & \mathtt{i386, PAE} \\ 512 & \mathtt{x86\_64} \end{cases}$$

pud_offset(pgd, addr) Receives as parameters a pointer pgd to a Page Global Directory entry and a linear address addr. The macro yields the linear address of the entry in a Page Upper Directory that corresponds to addr. In a two- or three-level paging system, this macro yields pgd, the address of a Page Global Directory entry.

Line 36 in pgtable-nopud.h:

```
static inline pud_t * pud_offset(pgd_t * pgd, unsigned long address)
{
        return (pud_t *)pgd;
}
```

pmd_offset(pud, addr) Receives as parameters a pointer pud to a Page Upper Directory entry and a linear address addr. The macro yields the address of the entry in a Page Middle Directory that corresponds to addr. In a two-level paging system, it yields pud, the address of a Page Global Directory entry.

Line 39 in pgtable-nopmd.h:

```
static inline pmd_t * pmd_offset(pud_t * pud, unsigned long address)
{
        return (pmd_t *)pud;
}
```

pte_offset_kernel(dir, addr) Yields the linear address of the Page Table that corresponds to the linear address addr mapped by the Page Middle Directory dir. Used only on the master kernel page tables (see the later section ”Kernel Page Tables”).

Line 335 in pgtable.h:

```
/*
 * the pte page can be thought of an array like this: pte_t[PTRS_PER_PTE]
 *
 * this macro returns the index of the entry in the pte page which would
 * control the given virtual address
 */
#define pte_index(address) \
                (((address) >> PAGE_SHIFT) & (PTRS_PER_PTE - 1))
#define pte_offset_kernel(dir, address) \
        ((pte_t *) pmd_page_kernel(*(dir)) +  pte_index(address))
```

pte_offset_map(dir, addr) Receives as parameters a pointer dir to a Page Middle Directory entry and a linear address addr; it yields the linear address of the entry in the Page Table that corresponds to the linear address addr. If the Page Table is kept in high memory, the kernel establishes a temporary kernel mapping (see the section ”Kernel Mappings of High-Memory Page Frames” in Chapter 8), to be released by means of pte_unmap. The macros pte_offset_map_nested and pte_unmap_nested are identical, but they use a different temporary kernel mapping.

Line 370 in pgtable.h:

```
#if defined(CONFIG_HIGHPTE)
#define pte_offset_map(dir, address) \
        ((pte_t *)kmap_atomic(pmd_page(*(dir)),KM_PTE0) + pte_index(address))
#define pte_offset_map_nested(dir, address) \
        ((pte_t *)kmap_atomic(pmd_page(*(dir)),KM_PTE1) + pte_index(address))
#define pte_unmap(pte) kunmap_atomic(pte, KM_PTE0)
#define pte_unmap_nested(pte) kunmap_atomic(pte, KM_PTE1)
#else
#define pte_offset_map(dir, address) \
        ((pte_t *)page_address(pmd_page(*(dir))) + pte_index(address))
#define pte_offset_map_nested(dir, address) pte_offset_map(dir, address)
#define pte_unmap(pte) do { } while (0)
#define pte_unmap_nested(pte) do { } while (0)
#endif
```

pte_none, pmd_none, pud_none, pgd_none yield the value 1 if the corresponding entry has the value 0; otherwise, they yield the value 0.

pmd_bad, pud_bad, pgd_bad The pmd_bad macro is used by functions to check Page Middle Directory entries passed as input parameters. It yields the value 1 if the entry points to a bad Page Table that is, if at least one of the following conditions applies:

- The page is not in main memory (Present flag cleared).
- The page allows only Read access (Read/Write flag cleared).
- Either Accessed or Dirty is cleared (Linux always forces these flags to be set for every existing Page Table).

The pud_bad and pgd_bad macros always yield 0. No pte_bad macro is defined, because it is legal for a Page Table entry to refer to a page that is not present in main memory, not writable, or not accessible at all.

5.3 Physical Memory Layout

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

```
  0xFFFFFFFF +---------------------+ 4GB
Reset vector |   JUMP to 0xF0000   |
  0xFFFFFFF0 +---------------------+ 4GB - 16B
             |    Unaddressable    |
             | memory, real mode   |
             | is limited to 1MB.  |
    0x100000 +---------------------+ 1MB
             |     System BIOS     |
     0xF0000 +---------------------+ 960KB
             |  Ext. System BIOS   |
     0xE0000 +---------------------+ 896KB
             |    Expansion Area   |
             | (maps ROMs for old  |
             |  peripheral cards)  |
     0xC0000 +---------------------+ 768KB
             | Legacy Video Card   |
             |    Memory Access    |
     0xA0000 +---------------------+ 640KB
             |    Accessible RAM   |
             |  (640KB is enough   |
             |  for anyone - old   |
             |      DOS area)      |
           0 +---------------------+ 0
```

Why isn't the kernel loaded starting with the first available megabyte of RAM? ([BC05, Sec 2.5.3]) Well, the PC architecture has several peculiarities that must be taken into account. For example:

- Page frame 0 is used by BIOS to store the system hardware configuration detected during the Power-On Self-Test(POST); the BIOS of many laptops, moreover, writes data on this page frame even after the system is initialized.

- Physical addresses ranging from `0x000a0000` to `0x000fffff` are usually reserved to BIOS routines and to map the internal memory of ISA graphics cards. This area is the well-known hole from 640 KB to 1 MB in all IBM-compatible PCs: the physical addresses exist but they are reserved, and the corresponding page frames cannot be used by the operating system.
- Additional page frames within the first megabyte may be reserved by specific computer models. For example, the IBM ThinkPad maps the 0xa0 page frame into the 0x9f one.

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`

Memory Initialization Steps ([Mau08, Sec 3.4])

```
setup_arch
 ├─ machine_specific_memory_setup
 ├─ parse_early_param
 ├─ setup_memory
 ├─ paging_init
 │   └─ pagetable_init
 └─ zone_sizes_init
     ├─ add_active_range
     └─ free_area_init_nodes
```

- `setup_arch` is invoked from within `start_kernel()`
- `machine_specific_memory_setup`: to create a list with the memory regions occupied by the system and the free memory regions
- `parse_early_param`: parsing commandline arguments like `mem=XXX[KkmM]`, `highmem=XXX[kKmM]`, or `memmap=XXX[KkmM]””@XXX[KkmM]` arguments
- `setup_memory`
  - The number of physical pages available (per node) is determined.
  - The bootmem allocator is initialized ([Mau08, Sec 3.4.3])
  - Various memory areas are then reserved, for instance, for the initial RAM disk needed when running the first userspace processes.
- `paging_init`: initializes the kernel page tables and enables paging
- `pagetable_init`: initializes the direct mapping of physical memory into the kernel address space. All page frames in low memory are directly mapped to the virtual memory region above `PAGE_OFFSET`. **This allows the kernel to address a good part of the available memory without having to deal with page tables anymore**.
- `zone_sizes_init`: initializes the `pgdat_t` instances of all nodes of the system

1. add_active_range: a comparatively simple list of the available physical memory is prepared
2. free_area_init_nodes: uses this information (got in above step) to prepare the full-blown kernel data structures

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
0xffff0000	0xffffffff	Reserved

- The *ACPI data* area stores information about the hardware devices of the system written by the BIOS in the POST phase; during the initialization phase, the kernel copies such information in a suitable kernel data structure, and then considers these page frames usable.
- The *ACPI NVS* area is mapped to ROM chips of the hardware devices, and hence cannot be used.
- The physical address range starting from 0xffff0000 is marked as reserved, because it is mapped by the hardware to the BIOS's ROM chip (see Appendix A).
- Notice that the BIOS may not provide information for some physical address ranges (in the table, the range is 0x000a0000 to 0x000effff). To be on the safe side, Linux assumes that such ranges are not usable.
- The kernel might not see all physical memory reported by the BIOS: for instance, the kernel can address only 4GB of RAM if it has not been compiled with PAE support, even if a larger amount of physical memory is actually available.
- kernlediy.com: Boot Memory Allocator
- *Memory management during the boot process*, [Mau08, Sec 3.4.3]

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

```
 page           160             256                                   768
frame: 0 1      0xa0            0x100                                0x300
       +-+-------+-------+-------+--------+-------------+------+-------+-
       | | avail | resvd | avail | kernel | Initialized | BSS  | avail |..
       | |       |       |       |  code  |    data     | data |       |..
       +-+-------+-------+-------+--------+-------------+------+-------+-
       0 4K     640K          _text    _etext        _edata   _end     3M
```

- You can find the linear address of these symbols (`_text`, `_etext`, `_edata`, `_end`) in the file `System.map`, which is created right after the kernel is compiled.

5.4 Process Page Tables

Process Page Tables

`0xC0000000` ⇔ `PAGE_OFFSET`

`include/asm-i386/page.h`

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

```
                  PAGE_OFFSET
+---------------------+-------------+
| user or kernel mode | kernel mode |
+---------------------+-------------+
0                     3G            4G
```

Why?

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

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

4G/4G solution

- (LWN article) There are users out there wanting to scale 32-bit Linux systems up to 32GB or more of main memory, so the enterprise-oriented Linux distributors have been scrambling to make that possible. One approach is the 4G/4G patch written by Ingo Molnar. This patch separates the kernel and user address spaces, allowing user processes to have 4GB of virtual memory while simultaneously expanding the kernel's low memory to 4GB. There is a cost, however: *the translation buffer (TLB) is no longer shared and must be flushed for every transition between kernel and user space*. Estimates of the magnitude of the performance hit vary greatly, but numbers as high as 30% have been thrown around. This option makes some systems work, however, so Red Hat ships a 4G/4G kernel with its enterprise offerings.

  Better solution: **go get a 64-bit system**.

- (LWN article: 4G/4G split on x86, 64 GB RAM (and more) support) Performance impact of the 4G/4G feature:

  There's a runtime cost with the 4G/4G patch: to implement separate address spaces for the kernel and userspace VM, the entry/exit code has to switch between the kernel pagetables and the user pagetables. This causes TLB flushes, which are quite expensive, not so much in terms of TLB misses (which are quite fast on Intel CPUs if they come from caches), but in terms of the direct TLB flushing cost (`cr3` manipulation) done on system-entry.

- (*Initialization of Paging*, [Mau08, Sec 3.4.2, p175]) It would also be possible to get rid of the split completely by introducing two 4 GiB address spaces, one for the kernel and one for each userspace program. However, context switches between kernel and user mode are more costly in this case.

Linux Memory Management Overview (a bit old)[tldp.org]

- A process' `PGDir` is initialized during a fork by `copy_page_tables()`. The idle process (`swapper`) has its `PGDir` initialized during the initialization sequence (`swapper_pg_dir`).

- The kernel code and data segments are priveleged segments defined in the global descriptor table (GDT) and extend from 3 GB to 4 GB. The swapper page directory (`swapper_pg_dir`) is set up so that logical addresses and physical addresses are identical in kernel space.

- Each user process has a local descriptor table (LDT) that contains a code segment and data-stack segment. These user segments extend from 0 to 3 GB (0xc0000000). In user space, linear addresses and logical addresses are identical.

- The space above 3 GB appears in a process' `PGDir` as pointers (each entry in `PGDir` has a pointer) to kernel page tables. (*The entries of the PGDir*, [BC05, Sec 2.5.4])

  This space is invisible to the process in user mode but the mapping becomes relevant when privileged mode is entered, for example, to handle a system call. Supervisor mode is entered within the context of the current process so address translation occurs with respect to the process' `PGDir` but using kernel segments. This is identically the mapping produced by using the `swapper_pg_dir` and kernel segments as both page directories use the same page tables in this space. Only `task[0]` (the idle task, sometimes called the swapper task for historical reasons, even though it has nothing to do with swapping in the Linux implementation) uses the `swapper_pg_dir` directly.

  - The user process' `segment_base = 0x00`, `page_dir` private to the process.
  - user process makes a system call: `segment_base=0xc0000000`, `page_dir = same user page_dir`.
  - `swapper_pg_dir` contains a mapping for all physical pages from `0xc0000000` to `0xc0000000 + end_mem`, so the first 768 entries in `swapper_pg_dir` are 0's, and then there are 4 or more that point to kernel page tables.
  - The user page directories have the same entries as `swapper_pg_dir` above 768. The first 768 entries map the user space.

- The upshot is that whenever the linear address is above `0xc0000000` everything uses the same kernel page tables.

- The user stack sits at the top of the user data segment and grows down. The kernel stack is not a pretty data structure or segment that I can point to with a "yon lies the kernel stack." A `kernel_stack_frame` (a page) is associated with each newly created process and is used whenever the kernel operates within the context of that process. Bad things would happen if the kernel stack were to grow below its current stack frame.

***The entries of the PGDir*, [BC05, Sec 2.5.4]**

- lower than 0xc0000000: (the first 768 entries with PAE disabled, or the first 3 entries with PAE enabled) depends on the specific process
  - Each user process thinks it has 3 GiB of memory
- higher than 0xc0000000: can be addressed only when the process is in kernel mode. This address space is common to all the processes and equal to the corresponding entries of the master kernel PGDir (see the following section)
- For 32-bit systems without PAE, PGDir
  - is 10-bit long
  - has 2^{10} (1K) entries
  - each PGDIR entry covers 2^{22} (4M)
  - the first 768 entries cover $768 \times 2^{22} = 3G$
- For 32-bit systems with PAE enabled, PGDir
  - is 2-bit long
  - has $2^2 = 4$ entries
  - each PGDIR entry covers $2^{30} = 1G$
  - the first 3 entries cover $3 \times 2^{30} = 3G$
- (*Significance of PAGE_OFFSET*, [NGC02, Sec 1.3.1]) The address space after PAGE_OFFSET is reserved for the kernel and this is where the complete physical memory is mapped (eg. if a system has 64mb of RAM, it is mapped from PAGE_OFFSET to PAGE_OFFSET + 64mb). This address space is also used to map non-continuous physical memory into continuous virtual memory.

5.5 Kernel Page Tables

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

```
      +---------------------+         4G +---------------------+
0x3FF |         0           |            :                     :
      :         0           :         8M |                     |
      +---------------------+            :                     :
0x301 |      -> pg1         |            +---------------------+
      +---------------------+            |       pg1 (4K)      |
0x300 |      -> pg0         |            +---------------------+
      +---------------------+            |       pg0 (4K)      |
      :         0           :            +---------------------+ _end
      +---------------------+            |                     |
    1 |      -> pg1         |            |    swapper_pg_dir   |
      +---------------------+            |---------------------|__bss_start
    0 |      -> pg0         |            |     Kernel image    |
      +---------------------+        +1M +---------------------+ _text
          swapper_pg_dir                 |     BIOS, holes...  |
                                      3G +---------------------+ PAGE_OFFSET
                                         :                     :
8M |                     |            8M |                     |
   :                     :               :                     :
   +---------------------+               +---------------------+
   |       pg1 (4K)      |               |       pg1 (4K)      |
   +---------------------+               +---------------------+
   |       pg0 (4K)      |               |       pg0 (4K)      |
   +---------------------+ _end          +---------------------+ _end
   |                     |               |                     |
   |    swapper_pg_dir   |               |    swapper_pg_dir   |
   |---------------------|__bss_start    |---------------------|__bss_start
   |     Kernel image    |               |     Kernel image    |
1M +---------------------+ _text      1M +---------------------+ _text
   |     BIOS, holes...  |               |     BIOS, holes...  |
 0 +---------------------+             0 +---------------------+
      Physical memory                     Kernel virtual memory
```

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
$0 \sim 8MB$	$\Rightarrow$	$0 \sim 8MB$
PAGE_OFFSET $\sim$ (PAGE_OFFSET $+ 8MB$)	$\nearrow$	

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

- `.fill 1024, 4, 0`[1] initializes a 4K area for `swapper_pg_dir`.
- `swapper_pg_dir` is at the beginning of BSS (uninitialized data area) because BSS is no longer used after system start up.
- assuming that the kernel's segments, the provisional Page Tables, and the 128KB memory area fit in the first 8 MB of RAM.
- In order to map 8 MB of RAM, two Page Tables are required
- `pg0` is right after the end of BSS (`_end`)

We won't bother mentioning the Page Upper Directories and Page Middle Directories anymore, because they are equated to Page Global Directory entries. PAE support is not enabled at this stage.

The objective of this first phase of paging is to allow these 8 MB of physical RAM to be easily addressed both in real mode and protected mode. Therefore, the kernel must create a mapping from both the linear addresses `0x00000000` through `0x007fffff` (8M) and the linear addresses `0xc0000000` through `0xc07fffff` (8M) into the physical addresses `0x00000000` through `0x007fffff`. In other words, the kernel during its first phase of initialization can address the first 8 MB of RAM by either linear addresses identical to the physical ones or 8 MB worth of linear addresses, starting from `0xc0000000`.

Why? (quoted from [NGC02, Sec 1.3.2])

- All pointers in the compiled kernel refer to addresses $> PAGE\_OFFSET$. That is, the kernel is linked under the assumption that its base address will be `start_text` (I think; I don't have the code on hand at the moment), which is defined to be $PAGE\_OFFSET + (some\ small\ constant,\ call\ it\ C)$.
- All the kernel bootstrap code (mostly real mode code) is linked assuming that its base address is $0 + C$.

`head.S` is part of the bootstrap code. It's running in protected mode with paging turned off, so all addresses are physical. In particular, the instruction pointer is fetching instructions based on physical address. The instruction that turns on paging (`movl %eax, %cr0`) is located, say, at some physical address A.

As soon as we set the paging bit in `cr0`, paging is enabled, and starting at the very next instruction, all addressing, including instruction fetches, pass through the address translation mechanism (page tables). IOW, all address are henceforth virtual. That means that

1. We must have valid page tables, and
2. Those tables must properly map the instruction pointer to the next instruction to be executed.

[1] `.fill REPEAT, SIZE, VALUE`: This emits `REPEAT` copies of `SIZE` bytes.

That next instruction is physically located at address `A+4` (the address immediately after the ”`movl %eax, %cr0`” instruction), but from the point of view of all the kernel code — which has been linked at `PAGE_OFFSET` — that instruction is located at virtual address `PAGE_OFFSET+(A+4)`. Turning on paging, however, does not magically change the value of EIP [2]. The CPU fetches the next instruction from ***virtual*** address `A+4`; that instruction is the beginning of a short sequence that effectively relocates the instruction pointer to point to the code at `PAGE_OFFSET+A+(something)`.

But since the CPU is, for those few instructions, fetching instructions based on physical addresses ***but having those instructions pass through address translation***, we must ensure that both the physical addresses and the virtual addresses are :

1. Valid virtual addresses, and
2. Point to the same code.

That means that at the very least, the initial page tables must

1. map virtual address `PAGE_OFFSET+(A+4)` to physical address `(A+4)`, and must
2. map virtual address `A+4` to physical address `A+4`.

This dual mapping for the first 8MB of physical RAM is exactly what the initial page tables accomplish. The 8MB initally mapped is more or less arbitrary. It’s certain that no bootable kernel will be greater than 8MB in size. The identity mapping is discarded when the MM system gets initialized.

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)
```

[2] The value of EIP is still physically $A + 4$, not $PAGE\_OFFSET + (A + 4)$ yet. But since paging is just enabled, CPU could pass $A + 4$ through address translation.

- The identity mapping is discarded when the MM system gets initialized.
- `page_pde_offset = (__PAGE_OFFSET » 20); /* 0xC00, the 3K point.*/`

  The `PGDir` is one page (4K) in size. It's divided into two parts:

  1. first 3K (768 entries) for user mode
  2. last 1K (256 entries) for kernel mode
- `$(pg0 - __PAGE_OFFSET)` yields the physical address of `pg0` since here it is a linear address. Same case for `$(swapper_pg_dir - __PAGE_OFFSET)`
- Registers:

  `%edi` address of each page table entry, i.e. `pg0[0]..pg0[1023]`, `pg1[0]..pg1[1023]`.

  `%edx` address of `swapper_pg_dir[0]`| and then to `swapper_pg_dir[1]`.

  `%ecx` has two uses

  1. contents of `swapper_pg_dir[0]`, `swapper_pg_dir[1]`, `swapper_pg_dir[768]`, `swapper_pg_dir[769]`.
  2. loop counter (1024 -> 0)

  `%eax` `7, 4k+7, 8k+7 ... 8M-4k+7` for 2k page table entries in `pg0` and `pg1` respectively.

  `%ebp` `= 128k + 7 + &pg0[1023]` in the first round of loop. Its value cannot be determined at coding time, because the address of `pg0` is not known until compile/link time.
- `stosl`: stores the contents of EAX at the address pointed by `EDI`, and increments `EDI`. Equivalent to:

  1. `movl %eax, (%edi)`
  2. `addl $4, %edi`
- `cmpl, jb`: if `%eax` < `%ebp`, jump to 10;

  - At the end of the 1st round of loop, the value of `%eax` is `4M-4k+7`, while the value of `%ebp` depends on the address of `pg0`.
- `INIT_MAP_BEYOND_END`: `128KB` (see `http://kerneldiy.com/blog/?p=201`)

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: $i \times 4M \sim (i+1) \times 4M - 1$
 *             - Kernel mode: $3G + i \times 4M \sim 3G + (i+1) \times 4M - 1$
 *  + Physical address range: $i \times 4M \sim (i+1) \times 4M - 1$
 */
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;
    }
  }
```

Source: http://www.eefocus.com/article/09-04/71517s.html

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

```
void __init paging_init(void)
{
#ifdef CONFIG_X86_PAE
  /* ... */
#endif

  pagetable_init();
  load_cr3(swapper_pg_dir);

#ifdef CONFIG_X86_PAE
  /* ... */
#endif

  __flush_tlb_all();
  kmap_init();
  zone_sizes_init();
}
```

Is PAE enabled in your kernel? try `grep PAE /boot/config-*`

- paging_init() without PAE:
  1. Invokes pagetable_init() to set up the Page Table entries properly.
  2. Writes the physical address of swapper_pg_dir in the cr3 control register.
  3. Invokes __flush_tlb_all() to invalidate all TLB entries.

---

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:

```
pgd = swapper_pg_dir + pgd_index(PAGE_OFFSET); /* 768 */
phys_addr = 0x00000000;
while (phys_addr < (max_low_pfn * PAGE_SIZE))
{
  pmd = one_md_table_init(pgd); /* returns pgd itself */
  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; /* 0x400000, 4M */
  ++pgd;
}
```

- *Final kernel Page Table when RAM size is less than 896 MB*, [BC05, Sec 2.5.5.2]
- This loop begins setting up valid PMD entries to point to. In the PAE case, pages are allocated with `alloc_bootmem_low_pages()` and the PGD is set appropriately. Without PAE, there is no middle directory, so it is just "folded" back onto the PGD to preserve the illusion of a 3-level pagetable.(*Page Table Management*, [Gor04, Appendix C, p224])
- `#define pgd_index(address) (((address) » PGDIR_SHIFT) & (PTRS_PER_PGD-1))`
  - `(PAGE_OFFSET » PGDIR_SHIFT) & (PTRS_PER_PGD-1)` ⇒
  - `(C0000000 » 22) & (1024 - 1) = 0x300 & 1023 = 768`
- `pgd` is a pointer initialized to the `pgd_t` corresponding to the beginning of the kernel portion of the linear address space. The lower 768 entries are left alone for user space.
- `one_md_table_init()` is described in the comments of the source code. And it's easy to trace the calls in it to get a clear idea.

  Line 55 in `mm/init.c`:

  ```
  /*
   * Creates a middle page table and puts a pointer to it in the
   * given global directory entry. This only returns the gd entry
   * in non-PAE compilation mode, since the middle layer is folded.
   */
  static pmd_t * __init one_md_table_init(pgd_t *pgd)
  {
    pud_t *pud;
    pmd_t *pmd_table;

  #ifdef CONFIG_X86_PAE
    ...
  #else
    pud = pud_offset(pgd, 0);
    pmd_table = pmd_offset(pud, 0);
  #endif

    return pmd_table;
  }
  ```
- `#define set_pmd(pmdptr, pmdval) (*(pmdptr) = (pmdval))`

When RAM Size Is Between 896MB ~ 4096MB

Physical memory zones:

```
|                       |
|                       |
|     ZONE_HIGHMEM      |
| (not directly mapped) |
|                       |
+-----------------------+
|      ZONE_NORMAL      |
|   (directly mapped)   |
|      896M - 16M       |
+-----------------------+
|       ZONE_DMA        |
|         16M           |
+-----------------------+
      Physical RAM
```

Direct mapping for ZONE_NORMAL:

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

ℹ

- To initialize the Page Global Directory, the kernel uses the same code as in the previous case.
- The __pa macro is used to convert a linear address starting from PAGE_OFFSET to the corresponding physical address, while the __va macro does the reverse.
- ZONE_DMA - Contains page frames of memory below 16 MB
- ZONE_NORMAL - Contains page frames of memory at and above 16 MB and below 896 MB
- ZONE_HIGHMEM - Contains page frames of memory at and above 896 MB

High Memory

```
+--------------+
|              |
|              |  dynamic mapping
| ZONE_HIGHMEM |<--.
|              |    \___    +--------------+ ---
|              |        `-->|     128M     |
+--------------+            +--------------+ kernel
|              |<---------->|              | space
|     896M     |<---------->|     896M     |  (1G)
|              |<---------->|              |
+--------------+  directly  +--------------+ ---
      RAM          mapped   |              |
                            | application  | user
                            | program code | space
                            | and data     | (3G)
                            ~              ~
                            |              |
                            +--------------+ ---
                             Virtual memory
```

- Coping with HighMemory
  - Memory above the physical address of 896MB are temporarily mapped into kernel virtual memory whenever the kernel needs to access that memory.
  - Data which the kernel frequently needs to access is allocated in the lower 896MB of memory (`ZONE_NORMAL`) and can be immediately accessed by the kernel (see Temporary mapping).
  - Data which the kernel only needs to access occasionally, including page cache, process memory and page tables, are preferentially allocated from `ZONE_HIGHMEM`.
  - The system can have additional physical memory zones to deal with devices that can only perform DMA to a limited amount of physical memory, `ZONE_DMA` and `ZONE_DMA32`.
  - Allocations and pageout pressure on the various memory zones need to be balanced (see Memory Balancing).

- (From comments in http://kerneltrap.org/node/2450) **The whole 1st GB of physical RAM is reserved for kernel use?** In fact, it is. The kernel has to have control over the whole memory. It's the kernel's job to allocate/deallocate RAM to processes.

  What the kernel does is, maps the entire available memory in its own space, so that it can access any memory region. It then gives out free pages to processes which need them.

  The userspace processes cannot of its own allocate a page to itself. It has to request the kernel to give it some memory area. Once the kernel has mapped a physical page in the process's space, it can use that extra memory.

- (*High Memory*, [Gor04, Sec 2.5]) To access physical memory between the range of 1GiB and 4GiB, the kernel temporarily maps pages from high memory into `ZONE_NORMAL` with `kmap()`.

  Why? (*Kernel Mappings of High-Memory Page Frames*, [BC05, Sec 8.1.6]) Page frames above the 896 MB boundary are not generally mapped in the 4th GiB of the kernel linear address spaces, so the kernel is unable to directly access them. This implies that each page allocator function that returns the linear address of the assigned page frame doesn't work for high-memory page frames, that is, for page frames in the `ZONE_HIGHMEM` memory zone.

  For instance, suppose that the kernel invoked `__get_free_pages(GFP_HIGHMEM,0)` to allocate a page frame in high memory. If the allocator assigned a page frame in high memory, `__get_free_pages()` cannot return its linear address because it doesn't exist (there is no mapping between physical frame and virtual page); thus, the function returns NULL. In turn, the kernel cannot use the page frame; even worse, the page frame cannot be released because the kernel has lost track of it.

  Linux designers had to find some way to allow the kernel to exploit all the available RAM, up to the 64 GB supported by PAE. The approach adopted is the following:

  - The allocation of high-memory page frames is done only through the `alloc_pages()` function and its `alloc_page()` shortcut. These functions do not return the linear address of the first allocated page frame, because if the page frame belongs to the high memory, such linear address simply does not exist. Instead, *the functions return the linear address of the page descriptor of the first allocated page frame* (*Page Descriptors*, [BC05, Sec 8.1.1]). These linear addresses always exist, because all page descriptors are allocated in low memory once and forever during the kernel initialization.

  - Page frames in high memory that do not have a linear address cannot be accessed by the kernel. Therefore, part of the last 128 MB of the kernel linear address space is dedicated to mapping high-memory page frames. Of course, this kind of mapping is temporary, otherwise only 128 MB of high memory would be accessible. Instead, by recycling linear addresses the whole high memory can be accessed, although at different times.

- More about high memory:
  - *Page Table Setup*, [NGC02, Sec 1.8]
  - Stackoverflow: How does the linux kernel manage less than 1GB physical memory?
  - `http://www.cs.usfca.edu/~cruse/cs635/lesson04.ppt`
  - `http://ilinuxkernel.com/?p=1013`

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:

```c
pgd_idx = pgd_index(PAGE_OFFSET); /* 3 */

/* the first 3 entries are for user space, and are pointing to the same empty_zero_page.*/
for (i=0; i<pgd_idx; i++)
  set_pgd(swapper_pg_dir + i, __pgd(__pa(empty_zero_page) + 0x001)); /* 0x001 == Present */

/* the 4th entry is for kernel space*/
pgd = swapper_pg_dir + pgd_idx;
phys_addr = 0x00000000;

/* i=3 initially. PTRS_PER_PGD=4 */
for (; i<PTRS_PER_PGD; ++i, ++pgd) {
  /* get the address of a PMD.
     The PMD maps 1G allocated by alloc_bootmem_low_pages() */
  pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
  /* the 4th entry is initialized with the above PMD */
  set_pgd(pgd, __pgd(__pa(pmd) | 0x001)); /* 0x001 == Present */

  if (phys_addr < max_low_pfn * PAGE_SIZE) /* cover ZONE_NORMAL */
    for (j=0; j < PTRS_PER_PMD /* 512 */
      && phys_addr < max_low_pfn*PAGE_SIZE; ++j) {
      /* fill up each PMD entry */
      set_pmd(pmd, __pmd(phys_addr | pgprot_val(__pgprot(0x1e3))));
      /* 0x1e3 == Present, Accessed, Dirty, Read/Write,
         Page Size, Global */

      /* each PMD entry covers 2M */
      phys_addr += PTRS_PER_PTE * PAGE_SIZE; /* 0x200000 */
    }
 }

/* The fourth Page Global Directory entry is then copied into the first entry, so as to
   mirror the mapping of the low physical memory in the first 896 MB of the linear address
   space. This mapping is required in order to complete the initialization of SMP systems:
   when it is no longer necessary, the kernel clears the corresponding page table entries
   by invoking the zap_low_mappings() function, as in the previous cases. */
swapper_pg_dir[0] = swapper_pg_dir[pgd_idx];
```

- *Final kernel Page Table when RAM size is more than 4096 MB*, [BC05, Sec 2.5.5.4]
- this code can be used for both PAE and no-PAE situation. That's why the `for` loop

  `for (; i<PTRS_PER_PGD; ++i, ++pgd)`

  is used when `i=3, PTRS_PER_PGD=4`.
- `pgd_index(address)` returns the index of a PGDir entry

  `#define pgd_index(address) (((address) » PGDIR_SHIFT) & (PTRS_PER_PGD-1))`

  ⇒ `(C0000000 » 30) & (4 - 1)` = 3

  When PAE is enabled, there are 4 entries in `PGDir`. The first 3 entries are for user linear address space. The 4th entry is for kernel space.
- `#define set_pgd(pgdptr, pgdval) set_pud((pud_t *)(pgdptr), (pud_t) pgdval )` puds are folded into pgds so this doesn't get actually called, but the define is needed for a generic inline function.

```
set_pgd(swapper_pg_dir + i, __pgd(__pa(empty_zero_page) + 0x001));
```

The kernel initializes the first three entries in the `PGDir` corresponding to the user linear address space with the address of an empty page (`empty_zero_page`).

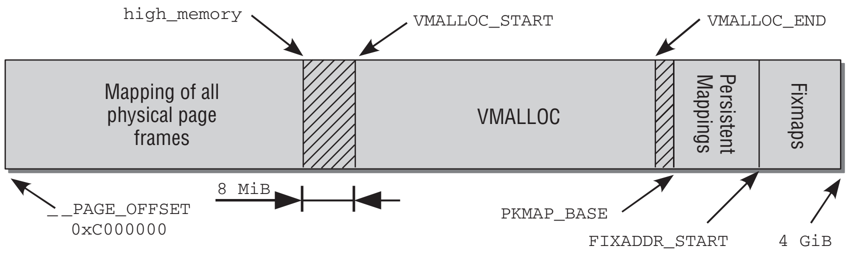
Line 417 in `arch/i386/kernel/head.S`:

```
ENTRY(empty_zero_page)
        .fill 4096,1,0
```

- The 4th entry is initialized with the address of a `pmd` allocated by invoking `alloc_bootmem_low_pages()`. There are 512 entries in a `pmd`,
  - the first 448 entries are filled with the physical address of the first 896MB of RAM.
  - the last 64 entries are reserved for noncontiguous memory allocation (*Noncontiguous Memory Area Management*, [BC05, Sec 8.3])

5.6 Fix-Mapped Linear Addresses

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.

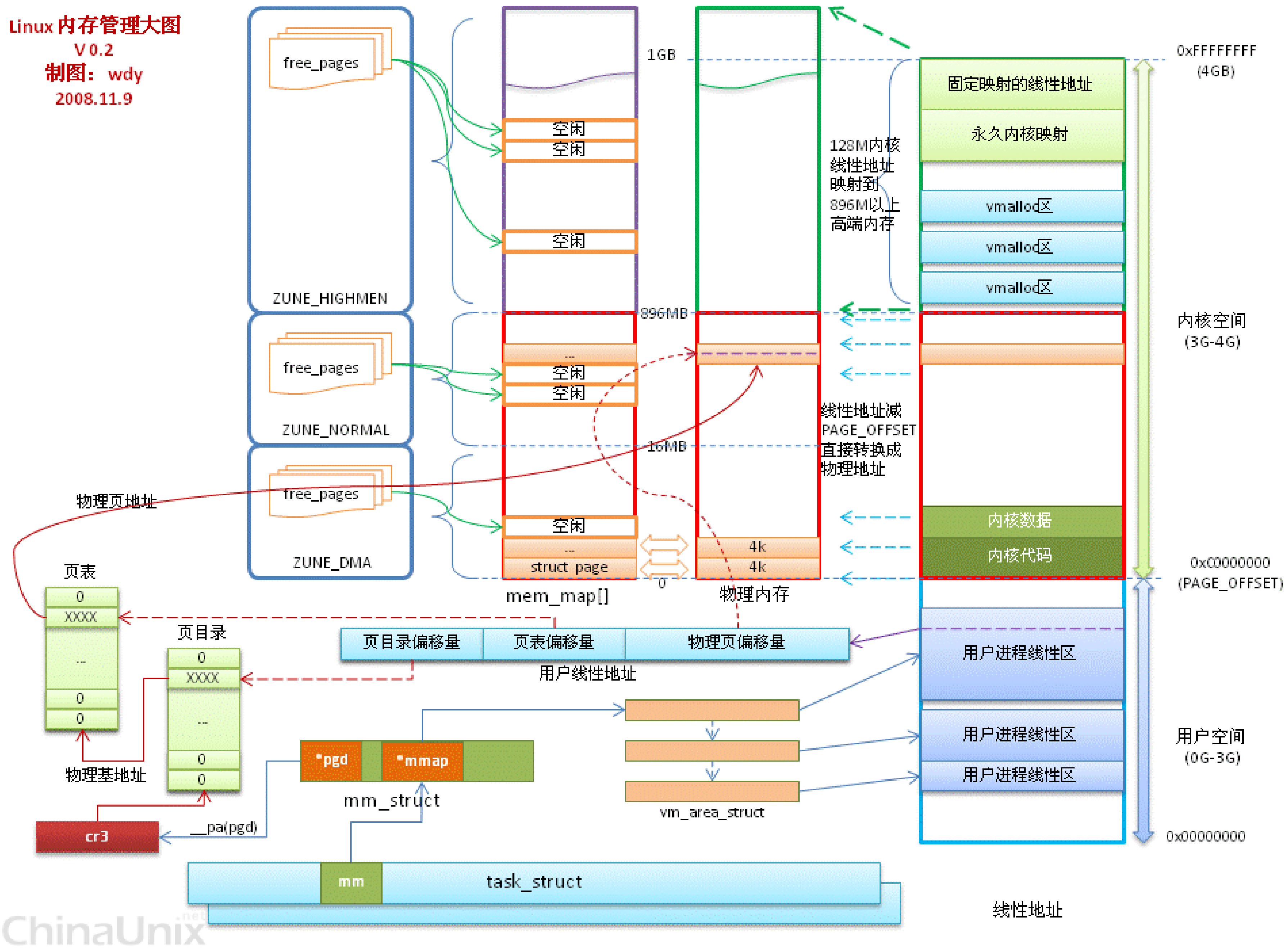
- *Architecture-Specific Setup*, [Mau08, Sec 3.4.2]

`VMALLOC_OFFSET` 8MB. (p178) This gap acts as a safeguard against any kernel faults. If out of bound addresses are accessed (these are unintentional accesses to memory areas that are no longer physically present), access fails and an exception is generated to report the error. If the vmalloc area were to immediately follow the direct mappings, access would be successful and the error would not be noticed. There should be no need for this additional safeguard in stable operation, but it is useful when developing new kernel features that are not yet mature.

Fixmaps (p180) The advantage of fixmap addresses is that at compilation time, the address acts like a constant whose physical address is assigned when the kernel is booted. Addresses of this kind can be de-referenced faster than when normal pointers are used. The kernel also ensures that the page table entries of fixmaps are not flushed from the TLB during a context switch so that access is always made via fast cache memory.

- *Fixmaps*, [NGC02, Sec 1.8.3]

Linux 内存管理大图
V 0.2
制图： wdy
2008.11.9


free_pages
ZUNE_HIGHMEN
free_pages
ZUNE_NORMAL
free_pages
ZUNE_DMA
空闲
空闲
空闲
...
空闲
空闲
空闲
...
struct page
mem_map[]
1GB
896MB
16MB
0
4k
4k
物理内存
128M内核
线性地址
映射到
896M以上
高端内存
线性地址减
PAGE_OFFSET
直接转换成
物理地址
固定映射的线性地址
永久内核映射
vmalloc区
vmalloc区
vmalloc区
内核数据
内核代码
用户进程线性区
用户进程线性区
用户进程线性区
线性地址
0xFFFFFFFF
(4GB)
内核空间
(3G-4G)
0xC0000000
(PAGE_OFFSET)
用户空间
(0G-3G)
0x00000000
物理页地址
页表
0
XXXX
...
0
0
页目录
0
XXXX
...
0
0
物理基地址
cr3
__pa(pgd)
*pgd
*mmap
mm_struct
页目录偏移量
页表偏移量
物理页偏移量
用户线性地址
vm_area_struct
mm
task_struct