

计算机系统基础
Programming Assignment

PA 3 存储管理

——PA 3-2 分段机制

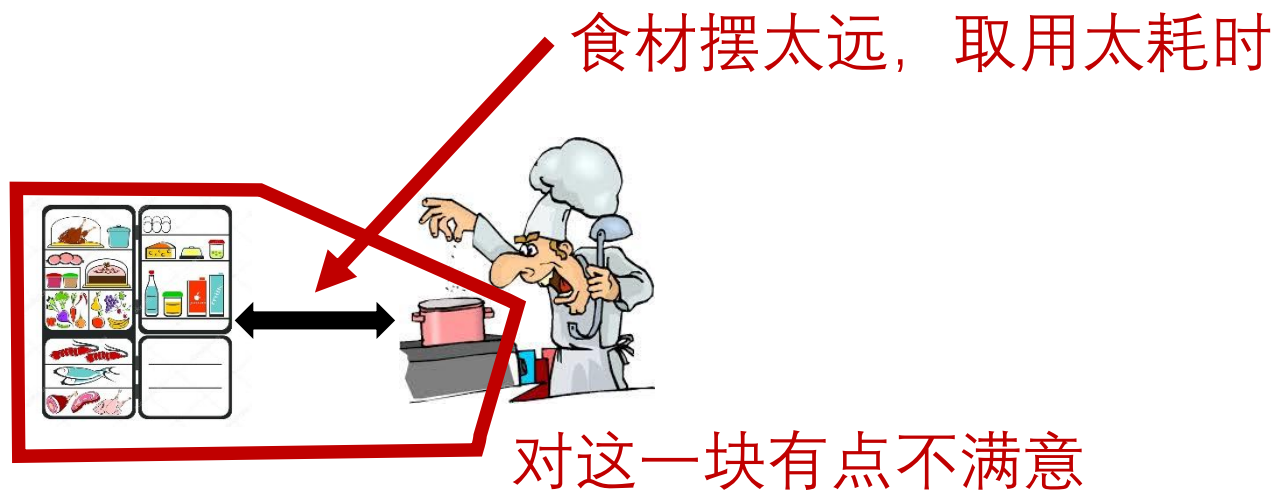
2017年12月1日

PA 3的总体任务 (以餐厅为类比)



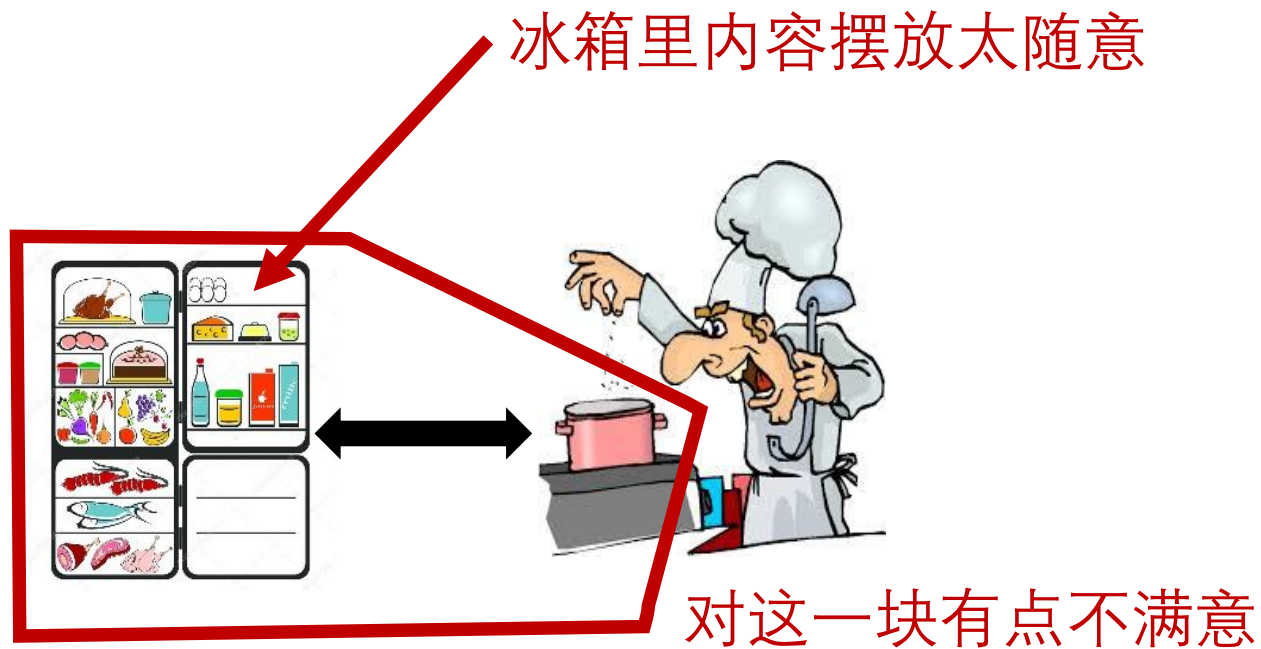
对这一块有点不满意

前期提要



- 于是在靠近处理器核心的地方
 - CPU的芯片上
- 放置了一些临时存储数据的单元
 - Cache
- 解决相应的设计问题
 - 问题一：Cache行和主存块的映射
 - 问题二：Cache中主存块的替换算法
 - 问题三：Cache一致性问题

分段机制的动机



分段机制的动机



乱

代码、数据都放在内存里，没有任何限制和保护。会导致很多问题：

1. 数组越界覆盖了代码
2. 恶意程序故意修改代码和数据

分段机制的动机



妥善组织，并提供保护



妥善组织

从第x格开始，连续n格，只能放熟食
从第y格开始，连续m格，只能放生食

.....

提供保护

熟食只能特级厨师才能存取
生食只要三级厨师就能存取

.....

分段机制的动机



分段机制

妥善组织，并提供保护



妥善组织

从地址 x 开始，连续 n 个字节，只能放代码（代码段）
从地址 y 开始，连续 m 个字节，只能放数据（数据段）
.....

提供保护

代码段只能等级0进程才能存取
数据段只要等级3进程就能存取
.....

如何保存分段信息？

冰箱上贴个条子

什么段	起始地址 (Base)	长度 (Limit)	权限要求 (DPL)
熟食 (代码)	0x0000	0x1111	特级 (0x0)
生食 (数据)	0x1111	0x1111	三级 (0x3)
蔬菜 (栈)	0x2222	0x1111	三级 (0x3)

如何保存分段信息？



在内存里就是一个叫做‘段表’的数组；除了32位的地址，额外用段寄存器存储‘什么段’这个信息；32位的地址变成段内偏移量。

什么段	起始地址 (Base)	长度 (Limit)	权限要求 (DPL)
熟食 (代码)	0x0000	0x1111	特级 (0x0)
生食 (数据)	0x1111	0x1111	三级 (0x3)
蔬菜 (栈)	0x2222	0x1111	三级 (0x3)

下面进入原理阶段

分段机制

- 现在NEMU的运行模式

```
uint32_t vaddr_read(vaddr_t vaddr, uint8_t sreg, size_t len) {  
    assert(len == 1 || len == 2 || len == 4);  
    return laddr_read(vaddr, len);  hwaddr_read(addr, len);  
}  
  
void vaddr_write(vaddr_t vaddr, uint8_t sreg, size_t len, uint32_t data) {  
    assert(len == 1 || len == 2 || len == 4);  
    laddr_write(vaddr, len, data);  hwaddr_write(addr, len, data);  
}
```

vaddr就是最终访问物理内存的物理地址
类似于**实模式**

分段机制

- 8086的实模式
 - 寄存器长度：16位
 - 包括段寄存器（segment register）：seg_reg
 - 地址线：20根
 - 物理地址计算方式
 - $\text{physical_address} = (\text{seg_reg} \ll 4) + \text{offset}$
 - 可寻址空间： $2^{20} = 1\text{MB}$
- NEMU类似但不同于实模式
 - 32位物理地址直接给出，不需任何转换

分段机制

- x86的机器开机后首先进入实模式
 - 加载操作系统
 - 操作系统初始化段表
 - 拨动一个‘开关’，从实模式切换到保护模式（开启分段机制）

分段机制

- x86的机器开机后首先进入实模式
 - 加载操作系统
 - 操作系统初始化段表
 - 拨动一个‘开关’，从实模式切换到保护模式（开启分段机制）
- 进入保护模式后
 - 程序给出48位逻辑地址（16位段选择符 + 32位有效地址）
 - 使用段选择符来查段表
 - 进行段级地址转换得到线性（现在就是物理）地址

分段机制

- x86的机器开机后首先进入实模式

- 加载操作系统

- 操作系统初始化段表

3

在代码阶段讲

- 1
- 拨动一个‘开关’，从实模式切换到保护模式（开启分段机制）

- 进入保护模式后

2

- 程序给出48位逻辑地址（16位段选择符 + 32位有效地址）

- 使用段选择符来查段表

- 进行段级地址转换得到线性（现在就是物理）地址

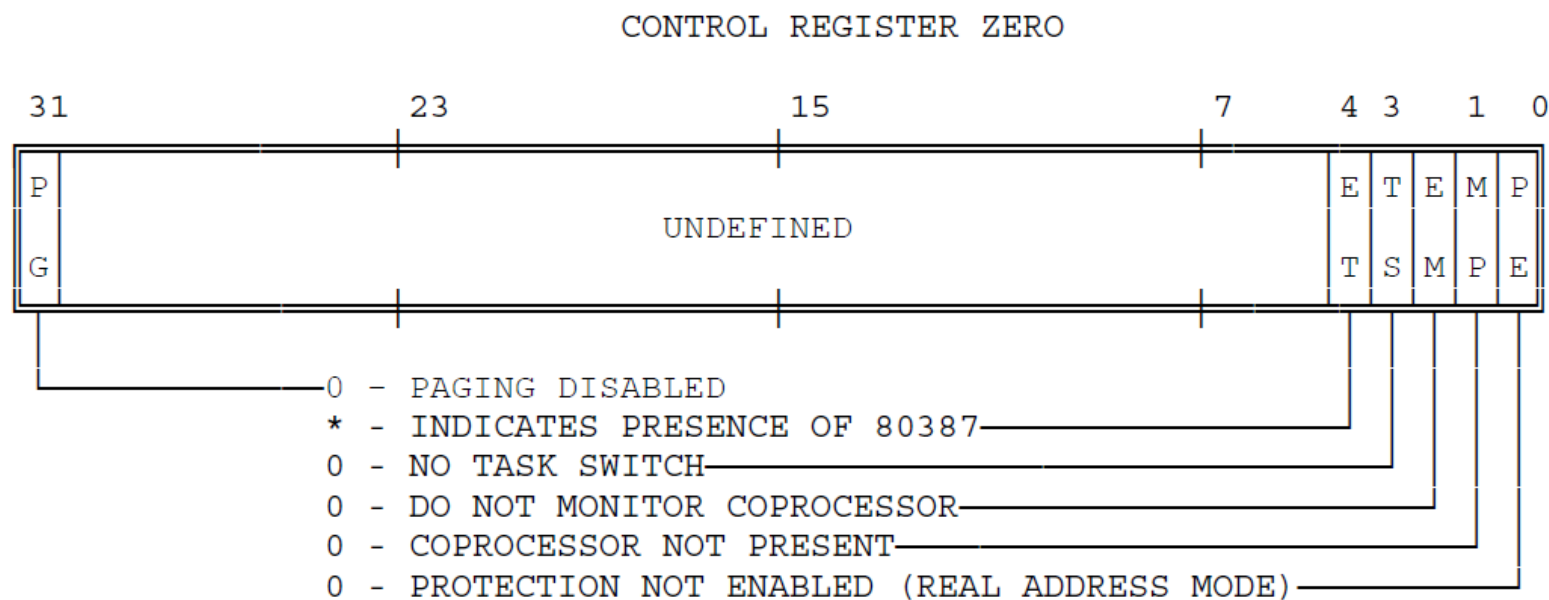
- PPT的讲述次序，注意和开机后的执行次序不同

分段机制（那个‘开关’）

- 80386的实模式和保护模式
 - 由CR0寄存器中的PE位控制

在NEMU中，CR0寄存器如何实现？

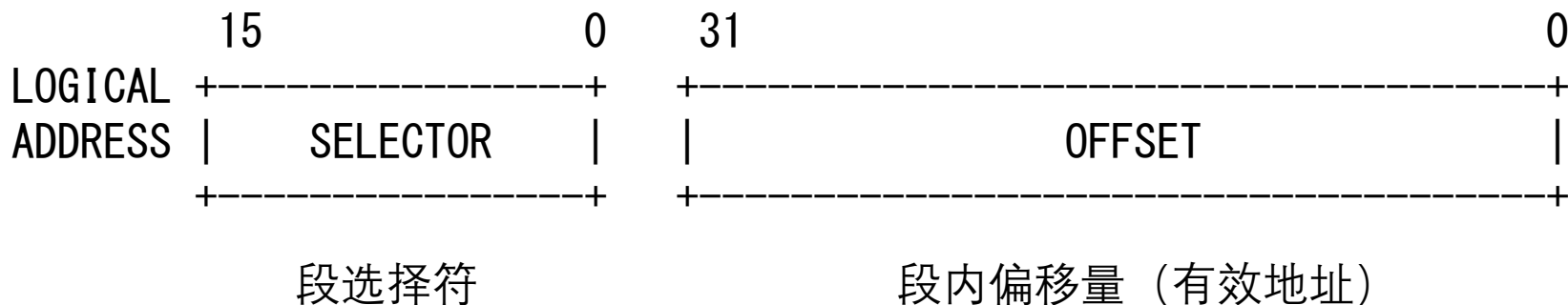
1. 参考EFLAGS寄存器（大小端序）
2. 是CPU_STATE的一个成员



当PE置为0时，采用实地址模式
当PE置为1时，采用保护地址模式

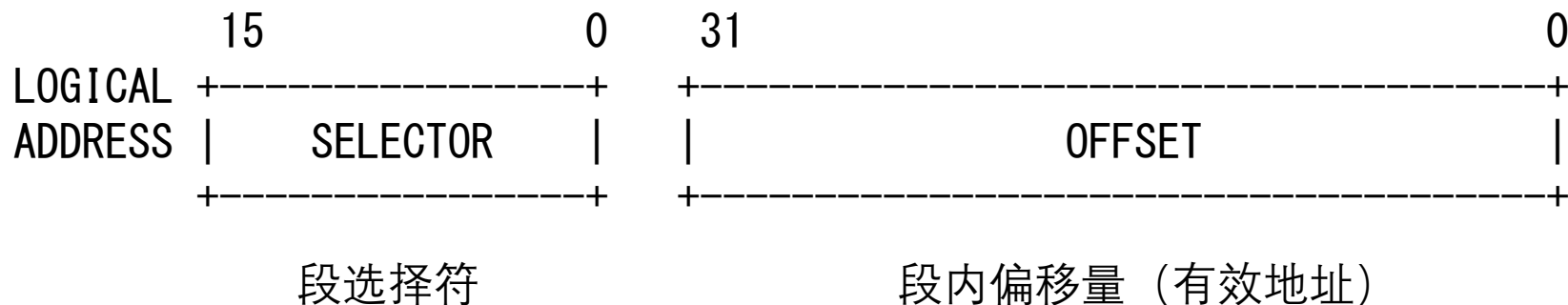
分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 逻辑地址：48位
 - 也称虚拟地址、虚地址
 - 其中
 - 段选择符：16位（sreg对应的段寄存器内容）
 - 段内偏移量（有效地址）：32位（vaddr给出的32位地址）



分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - CS, SS, DS, ES, FS, GS
 - CS：代码段寄存器
 - SS：栈段寄存器
 - DS：数据段寄存器
 - 其他三个：可以指向任意的数据段



分段机制 (地址转换)

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符

在NEMU中如何实现？

1. 参考EFLAGS
2. 是CPU STATE的一个成员



TI - TABLE INDICATOR

RPL – REQUESTOR'S PRIVILEGE LEVEL

TI :

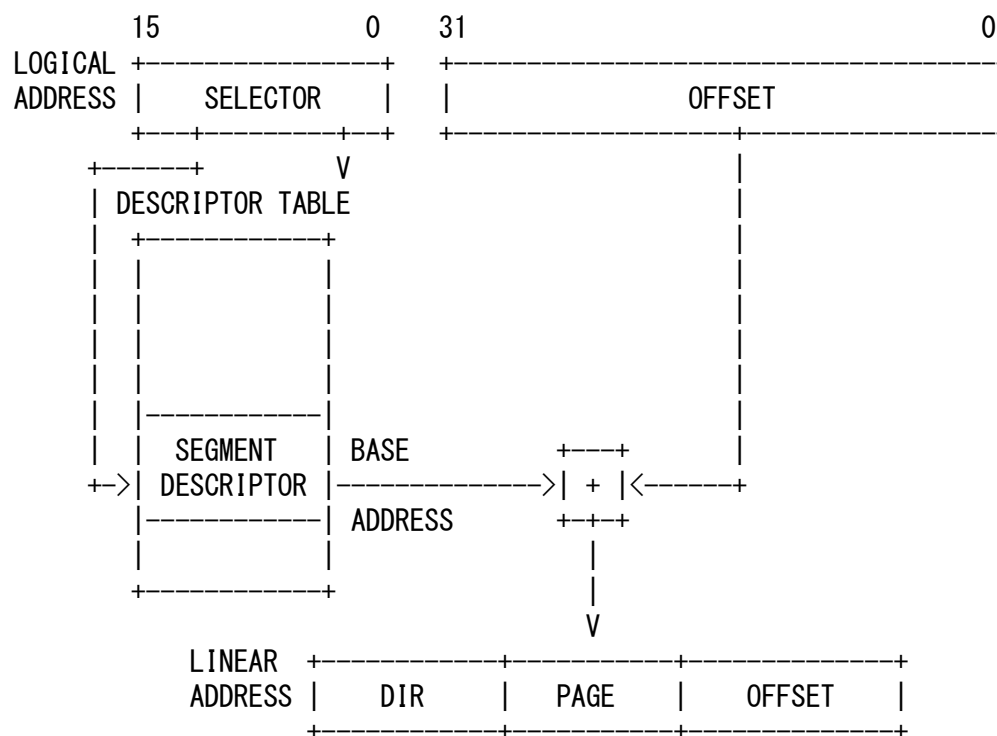
为0表示选择全局描述符表 (GDT)
为1表示选择局部描述符表 (LDT)

RPL：定义当前程序段的特权等级
00表示最高级，内核态
11表示最低级，用户态

Index：在段描述符表中的索引

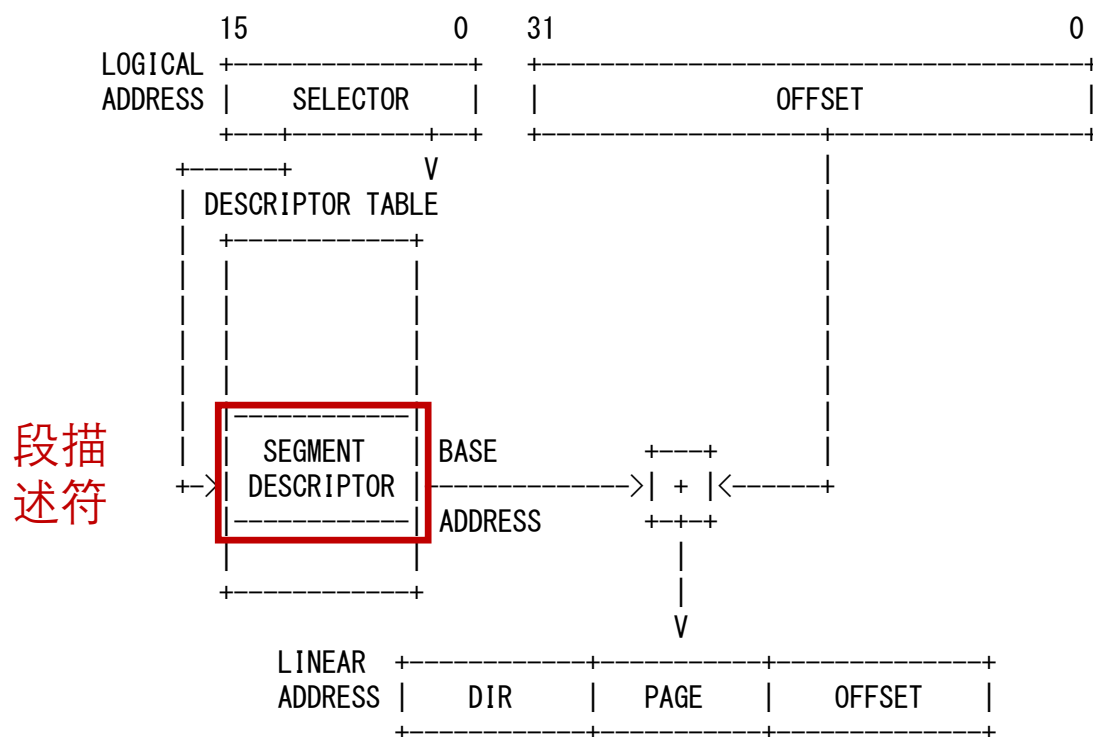
分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符



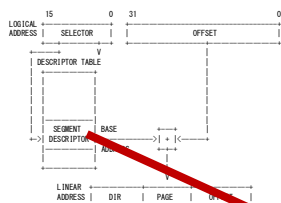
分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符



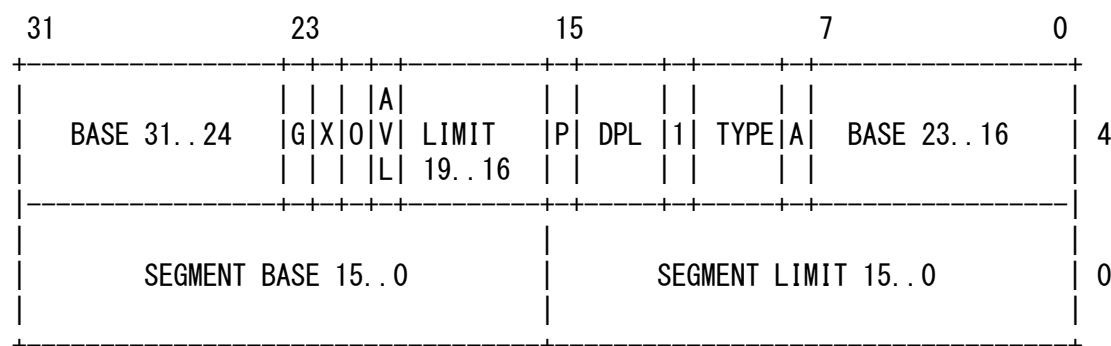
分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位



段描述符

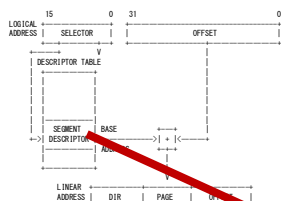
DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS



- A - ACCESSED
- AVL - AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
- DPL - DESCRIPTOR PRIVILEGE LEVEL
- G - GRANULARITY
- P - SEGMENT PRESENT

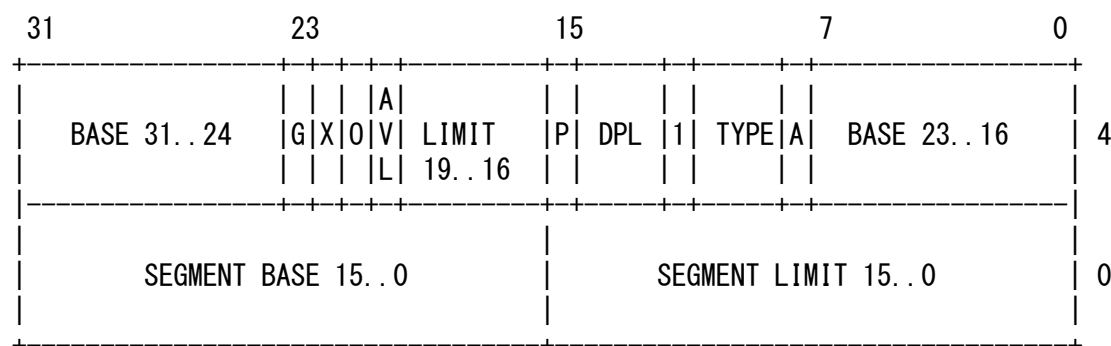
分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)



段描述符

DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS

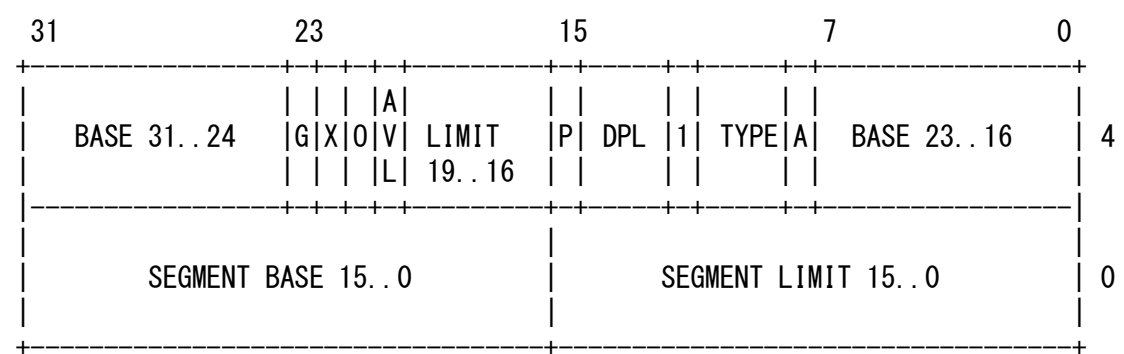


A - ACCESSED
AVL - AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
DPL - DESCRIPTOR PRIVILEGE LEVEL
G - GRANULARITY
P - SEGMENT PRESENT

分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)

DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS



- A - ACCESSED
- AVL - AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
- DPL - DESCRIPTOR PRIVILEGE LEVEL
- G - GRANULARITY
- P - SEGMENT PRESENT

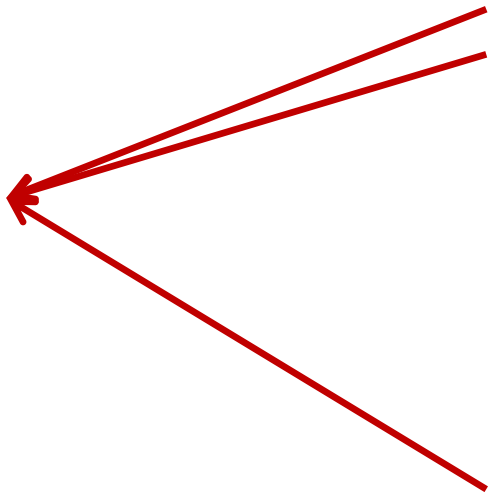
```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type           : 4;
    uint32_t segment_type   : 1;
    uint32_t privilege_level : 2;
    uint32_t present        : 1;
    uint32_t limit_19_16    : 4;
    uint32_t soft_use       : 1;
    uint32_t operation_size : 1;
    uint32_t pad0           : 1;
    uint32_t granularity    : 1;
    uint32_t base_31_24     : 8;
} SegDesc;
```

分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)

32位基地址

```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 1;
    uint32_t limit_19_16    : 4;
    uint32_t soft_use        : 1;
    uint32_t operation_size  : 1;
    uint32_t pad0            : 1;
    uint32_t granularity     : 1;
    uint32_t base_31_24     : 8;
} SegDesc;
```



分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)

```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 1;
    uint32_t limit_19_16    : 4;
    uint32_t soft_use        : 1;
    uint32_t operation_size  : 1;
    uint32_t pad0            : 1;
    uint32_t granularity     : 1;
    uint32_t base_31_24     : 8;
} SegDesc;
```

32位基地址

32位线性地址 =
32位基地址 +
32位段内偏移量（有效地址）

分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)

```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 1;
    uint32_t limit_19_16    : 4;
    uint32_t soft_use        : 1;
    uint32_t operation_size  : 1;
    uint32_t pad0            : 1;
    uint32_t granularity     : 1;
    uint32_t base_31_24     : 8;
} SegDesc;
```

20位界限


指出段的长度，用于检查地址越界，及偏移量超出最大段长的情况

分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)

粒度大小：为1表
示段以页（4KB）
为基本单位，为0
表示以字节为基
本单位

```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 1;
    uint32_t limit_19_16    : 4;
    uint32_t soft_use        : 1;
    uint32_t operation_size  : 1;
    uint32_t pad0            : 1;
    uint32_t granularity     : 1;
    uint32_t base_31_24     : 8;
} SegDesc;
```




分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)

相对于20位的界限，若
G = 0，则最大段长为？
G = 1，则最大段长为？

粒度大小：为1表
示段以页（4KB）
为基本单位，为0
表示以字节为基
本单位

```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 1;
    uint32_t limit_19_16    : 4;
    uint32_t soft_use        : 1;
    uint32_t operation_size  : 1;
    uint32_t pad0            : 1;
    uint32_t granularity     : 1;
    uint32_t base_31_24     : 8;
} SegDesc;
```



分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)

相对于20位的界限，若

G = 0，则最大段长为？ $2^{20}\text{B} = 1\text{MB}$

G = 1，则最大段长为？ $2^{20} * 4\text{KB} = 4\text{GB}$

粒度大小：为1表

示段以页（4KB）

为基本单位，为0

表示以字节为基

本单位

```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 1;
    uint32_t limit_19_16    : 4;
    uint32_t soft_use        : 1;
    uint32_t operation_size  : 1;
    uint32_t pad0            : 1;
    uint32_t granularity     : 1;
    uint32_t base_31_24     : 8;
} SegDesc;
```

分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)

存在位：为1表示段
已在内存中，为0表
示段不在内存中

```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 1;
    uint32_t limit_19_16    : 4;
    uint32_t soft_use        : 1;
    uint32_t operation_size  : 1;
    uint32_t pad0            : 1;
    uint32_t granularity     : 1;
    uint32_t base_31_24     : 8;
} SegDesc;
```

分段机制（地址转换）

- 80386保护模式下的地址转换

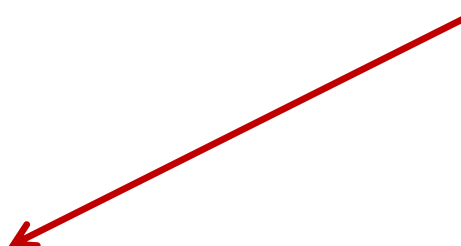
- 逻辑地址到线性地址的转换

- 段选择符存放在16位的段寄存器中：指向某段描述符

- 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)

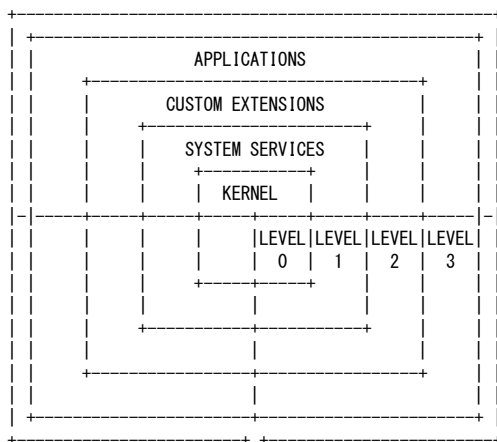
```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 1;
    uint32_t limit_19_16    : 4;
    uint32_t soft_use        : 1;
    uint32_t operation_size  : 1;
    uint32_t pad0            : 1;
    uint32_t granularity     : 1;
    uint32_t base_31_24     : 8;
} SegDesc;
```

特权级：访问段
时对当前特权级
的最低要求



分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)



特权级：访问段
时对当前特权级
的最低要求

```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 1;
    uint32_t limit_19_16    : 4;
    uint32_t soft_use        : 1;
    uint32_t operation_size  : 1;
    uint32_t pad0            : 1;
    uint32_t granularity     : 1;
    uint32_t base_31_24     : 8;
} SegDesc;
```

分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 段选择符存放在16位的段寄存器中：指向某段描述符
 - 段描述符为64位：[nemu/include/memory/mmu/segment.h](#)

只有当从数值上：

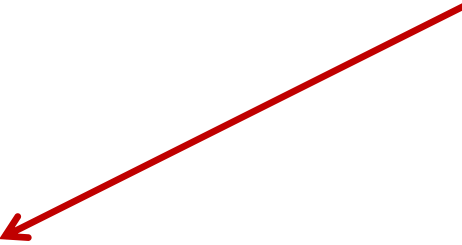
段描述符的DPL \geq 段选择符的RPL

段描述符的DPL \geq 进程的CPL

才有权访问该段

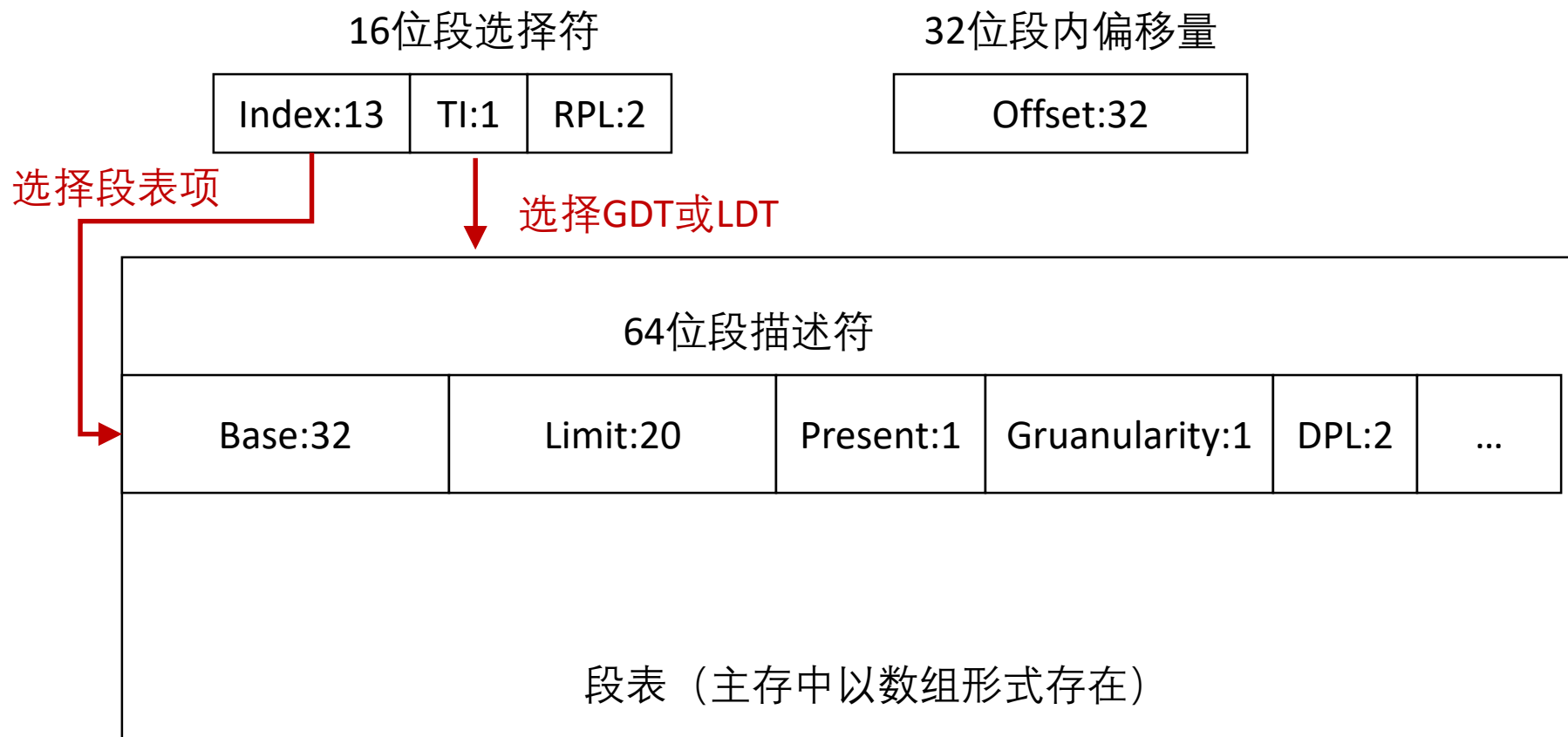
特权级：访问段
时对当前特权级
的最低要求

```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 1;
    uint32_t limit_19_16    : 4;
    uint32_t soft_use        : 1;
    uint32_t operation_size  : 1;
    uint32_t pad0            : 1;
    uint32_t granularity    : 1;
    uint32_t base_31_24     : 8;
} SegDesc;
```



分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换



分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换

16位段选择符

Index:13	TI:1	RPL:2
----------	------	-------

32位段内偏移量

Offset:32

64位段描述符

Base:32	Limit:20	Present:1	Gruanularity:1	DPL:2	...
---------	----------	-----------	----------------	-------	-----

检查缺段

段表（主存中以数组形式存在）

分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换

16位段选择符

Index:13	TI:1	RPL:2
----------	------	-------

32位段内偏移量

Offset:32

64位段描述符

Base:32	Limit:20	Present:1	Gruanularity:1	DPL:2	...
---------	----------	-----------	----------------	-------	-----

检查越界

段表（主存中以数组形式存在）

分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换

16位段选择符

Index:13	Tl:1	RPL:2
----------	------	-------

32位段内偏移量

Offset:32

进程CPL

64位段描述符

Base:32	Limit:20	Present:1	Gruanularity:1	DPL:2	...
---------	----------	-----------	----------------	-------	-----

检查访问权限

段表（主存中以数组形式存在）

分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换

16位段选择符

Index:13	TI:1	RPL:2
----------	------	-------

32位段内偏移量

Offset:32

64位段描述符

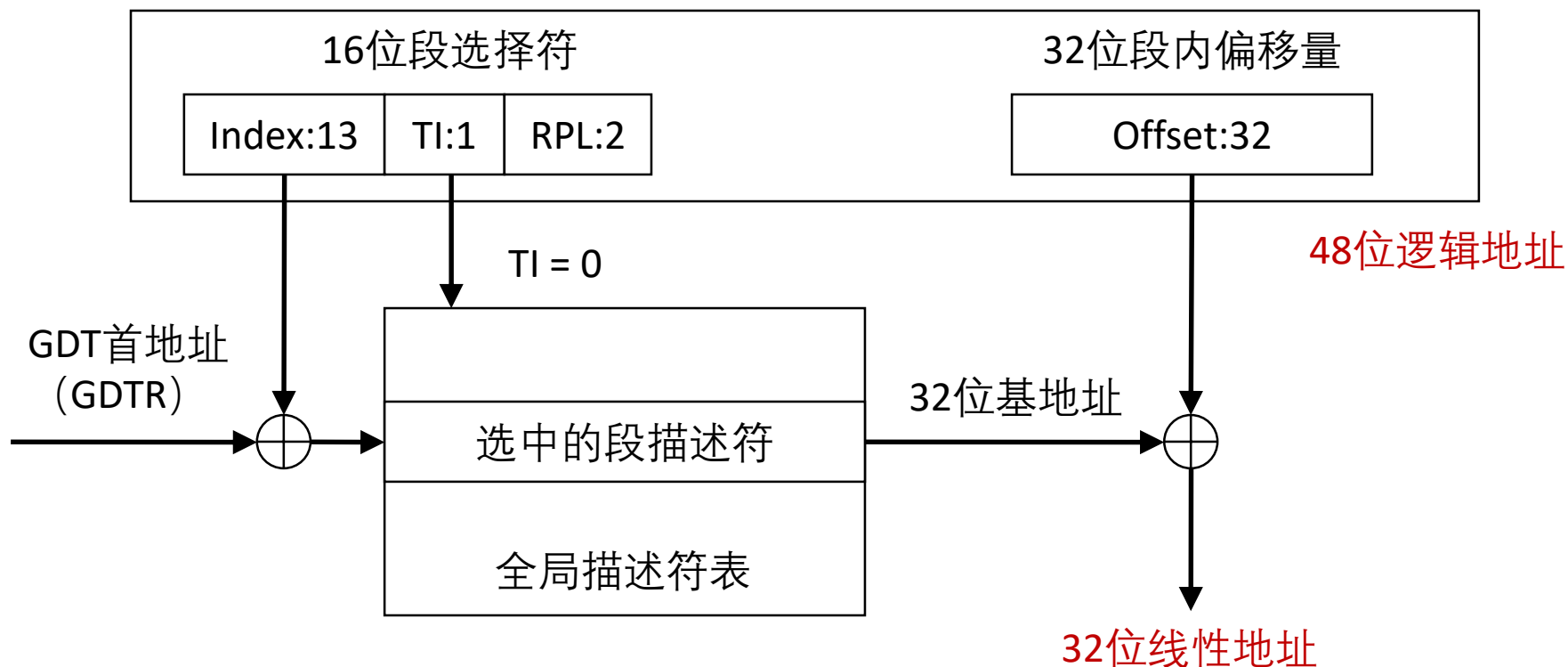
Base:32	Limit:20	Present:1	Gruanularity:1	DPL:2	...
---------	----------	-----------	----------------	-------	-----

线性地址=Base+Offset

段表（主存中以数组形式存在）

分段机制（地址转换）

- 80386保护模式下的地址转换
 - 逻辑地址到线性地址的转换
 - 如何找到GDT（LDT我们不会用到）
 - GDT的首地址（线性地址）存放于GDTR寄存器中



分段机制（地址转换）

- x86的机器开机后首先进入实模式
 - 加载操作系统
 - 操作系统初始化段表
 - 拨动一个‘开关’，从实模式切换到保护模式（开启分段机制）
- 进入保护模式后
 - 程序给出48位逻辑地址（16位段选择符 + 32位有效地址）
 - 使用段选择符来查段表
 - 进行段级地址转换得到线性（现在就是物理）地址
 - 根据段选择符找到段描述符
 - 根据段描述符找到段基地址
 - 线性地址 = 段基地址 + 段内偏移量（有效地址）
 - 检查：缺段、地址越界、访问权限

下面进入代码阶段

分段机制（开启分段机制）

- 在include/config.h中开启分段机制模拟
 - #define IA32_SEG
 - 引起kernel行为变化
 - 引起NEMU的行为变化

分段机制 (Kernel初始化段表)

```
#ifndef IA32_SEG
    ... // abandoned
#else

#define GDT_ENTRY(n) ((n) << 3)
#define MAKE_NULL_SEG_DESC \
#define MAKE_SEG_DESC(type,base,lim) \ ...
.globl start
start:
    ...//something about interrupt
    lgdt  va_to_pa(gdtdesc)
    movl  %cr0, %eax
    orl   $0x1, %eax
    movl  %eax, %cr0
    ljmp  $GDT_ENTRY(1), $va_to_pa(start_cond)
start_cond:
    # Set up the protected-mode data segment registers
    movw  $GDT_ENTRY(2), %ax
    movw  %ax, %ds      # %DS = %AX
    ...
    # Set up a stack for C code.
    ...      jmp init  # never return
```

这里在干啥？

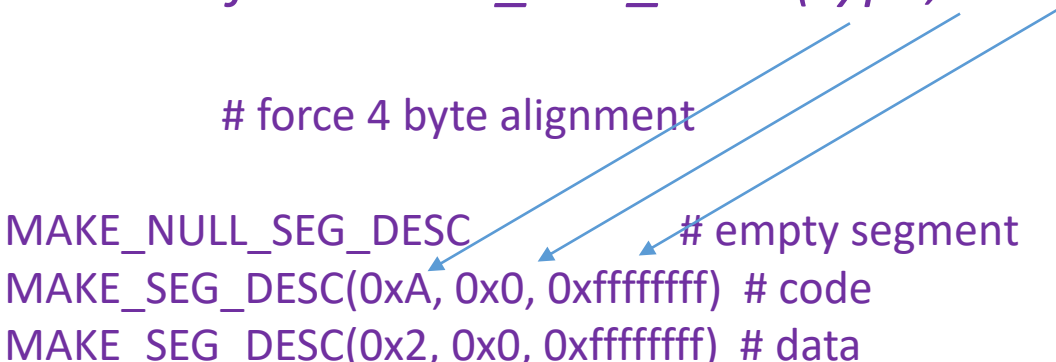
#这里在干啥？

#这里在干啥？

分段机制 (Kernel初始化段表)

接上一页

```
#define MAKE_SEG_DESC(type,base,lim) \ ...  
  
# GDT  
.p2align 2          # force 4 byte alignment  
gdt:  
    MAKE_NULL_SEG_DESC  
    MAKE_SEG_DESC(0xA, 0x0, 0xffffffff) # code  
    MAKE_SEG_DESC(0x2, 0x0, 0xffffffff) # data  
  
gdtdesc:            # descriptor  
    .word (gdtdesc - gdt - 1) # limit = sizeof(gdt) - 1  
    .long va_to_pa(gdt)      # address of GDT  
  
# end of IA32_SEG  
#endif
```



NEMU在什么时候，由谁指挥进入了保护模式？
扁平模式怎么体现？

分段机制（NEMU功能升级）

- NEMU中开启保护模式
 - CPU_state结构中添加相应寄存器
 - GDTR
 - CR0
 - 各段寄存器
 - 16bit visible selector
 - Hidden descriptor（隐藏部分）
 - I386手册5.1.4节，课本pg. 274图6.38
 - init_cpu()函数中初始化CR0和GDTR寄存器
 - 首先工作在实地址模式下

分段机制（NEMU功能升级）

- NEMU中开启保护模式
 - 添加lgdt指令
 - 查阅i386手册
 - 只在操作系统代码中出现

分段机制（NEMU功能升级）

- NEMU中开启保护模式
 - 添加`jmp`指令
 - 添加操作码为0F 20的`mov`指令
 - 添加操作码为0F 22的`mov`指令
 - 查阅i386手册
 - OPERAND的读写接口准备好了相应的功能
 - 在指令实现中调用`load_sreg()`把段表的base和limit等信息装载到段寄存器隐藏部分
 - `load_sreg()`定义在`nemu/src/memory/mmu/segment.h`
 - 读段表，装载必要信息到sreg的隐藏部分，隐藏部分结构参照guide、i386手册和课本
 - 作必要的检查，如`present == 1`, `granularity == 1`等

分段机制（NEMU功能升级）

- ModR/M与SIB字节解码代码中有关段寄存器的处理应该能够看懂了（框架代码已经做好了）
- OPERAND读写时建议对sreg部分的取值做检查，不然遇到奇怪bug不好调试
- 没有用框架代码写的指令，在访存时要注意正确设置opr -> sreg
 - sreg的值通过operand_read()/write()传入vaddr_read()/write()
 - vaddr_read()/write()根据条件进行段级地址转换
 - 详见下一页

分段机制 (NEMU功能升级)

- NEMU中开启保护模式
 - 修改vaddr_read()与vaddr_write()函数

```
uint32_t vaddr_read(vaddr_t vaddr, uint8_t sreg, size_t len) {  
    assert(len == 1 || len == 2 || len == 4);  
    #ifndef IA32_SEG  
        return laddr_read(vaddr, len);  
    #else  
        uint32_t laddr = vaddr;  
        if( /* what condition??? */ ) {  
            laddr = segment_translate(vaddr, sreg);  
        }  
        return laddr_read(laddr, len);  
    #endif  
}
```

segment_translate()读取sreg的
隐藏部分来获取base和limit,
该函数和load_sreg()都定义在
segment.c中

分段机制（勘误）

- NEMU中开启保护模式
 - 添加操作码为8E的mov指令
 - 查阅i386手册
 - 勘误：Page 345 of 421
 - - 8D /r MOV Sreg,r/m16 2/5,pm=18/19 Move r/m word to segment register
 - + 8E /r MOV Sreg,r/m16 2/5,pm=18/19 Move r/m word to segment register

总结

- Include/config.h中定义#define IA32_SET
- Kernel初始化行为改变
 - 初始化段表和GDTR
 - 打开CR0中PE位的开关
 - 装填各段寄存器（同时使用load_sreg()填入隐藏部分）
- NEMU访存行为改变
 - 开机初始化CR0和GDTR，配置为实模式
 - 操作数地址从32物理地址变为48位逻辑地址
 - vaddr_read()/write()时要根据条件判断是否进行段级地址转换
 - 转换得到的线性地址目前阶段作为物理地址使用

PA 3-2截止时间

2014年12月14日24时， 即， 15日0时

PA 3-2到此结束

祝大家学习快乐， 身心健康！

欢迎大家踊跃参加问卷调查

(高兴的话这阶段也可以填写量表)