

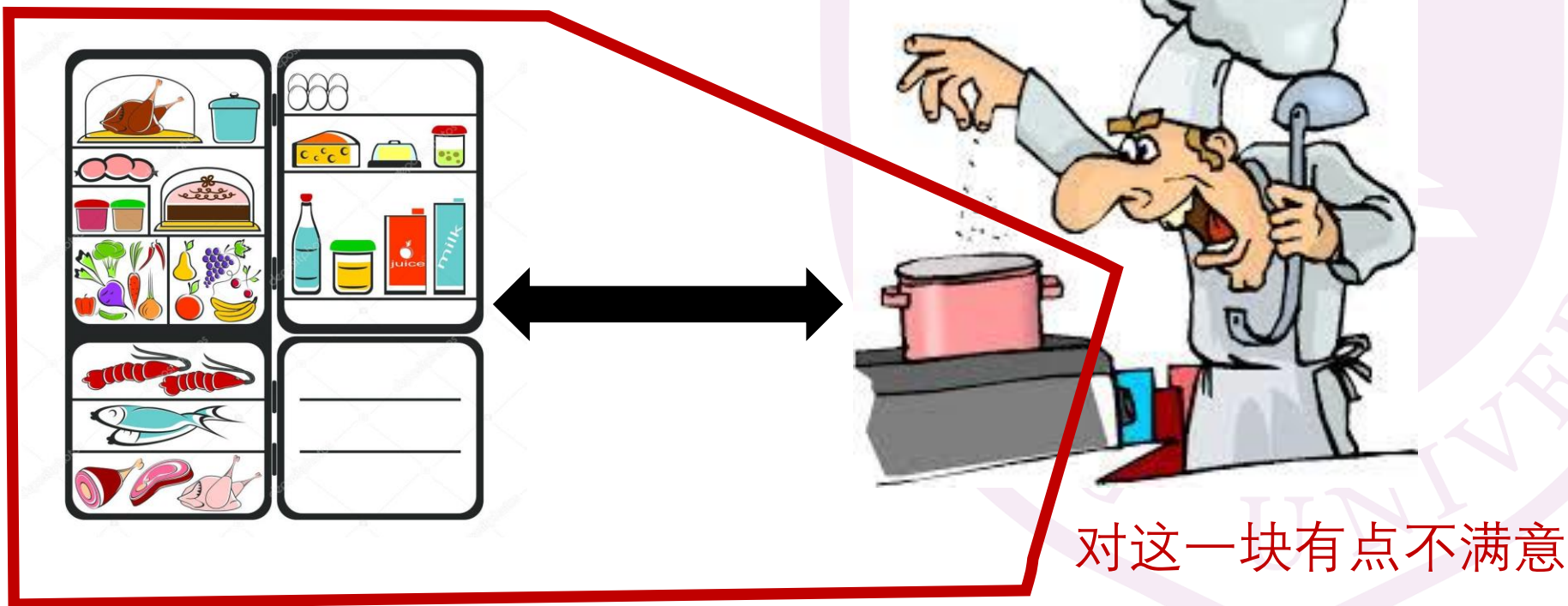
计算机系统基础
Programming Assignment

PA 3-2 分段机制的模拟

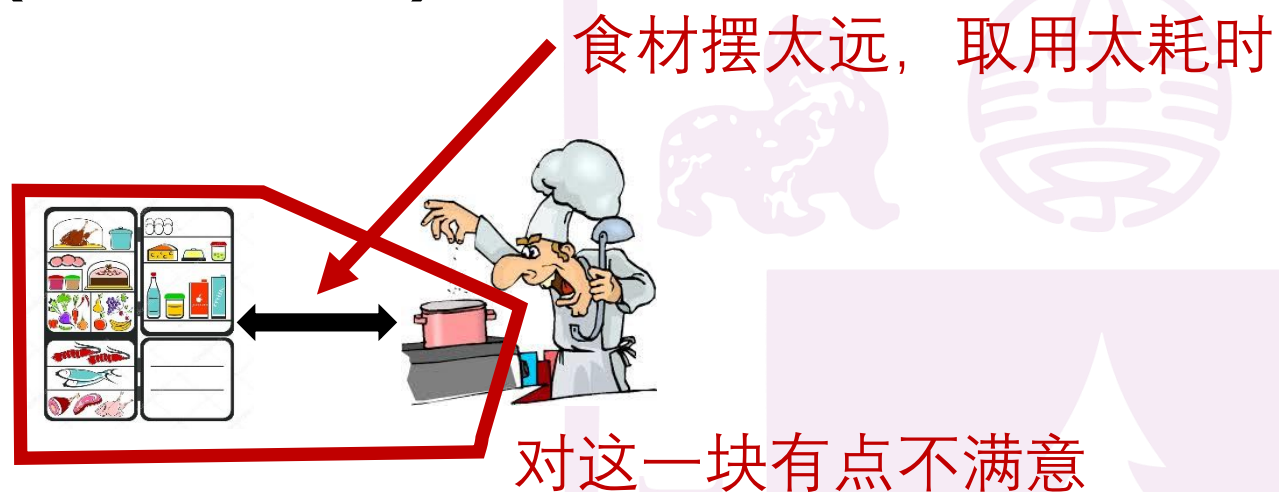
2020年12月3日 / 12月4日

南京大学《计算机系统基础》课程组

PA 3的总体任务 (以餐厅为类比)



前情提要 (PA 3-1)

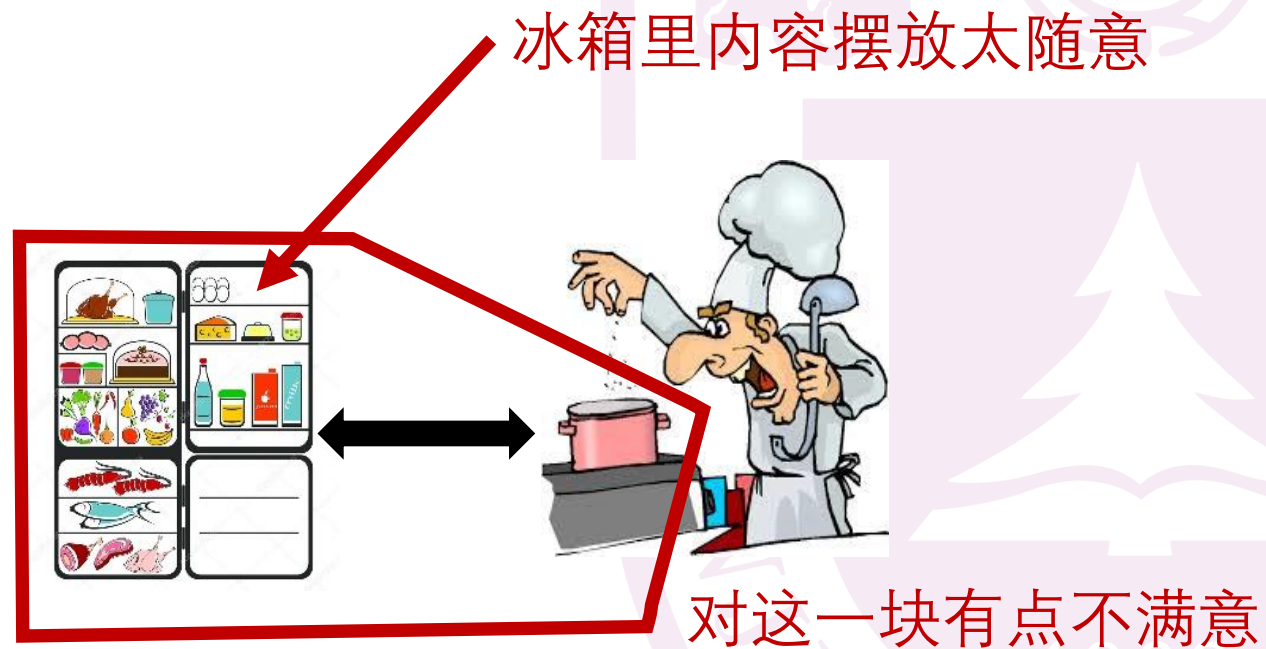


- 于是在靠近处理器核心的地方
 - CPU的芯片上
- 放置了一些临时存储数据的单元
 - Cache
- 解决相应的设计问题
 - 问题一: Cache行和主存块的映射
 - 问题二: Cache中主存块的替换算法
 - 问题三: Cache一致性问题



对内存进行分段保护的动机和基本思想

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

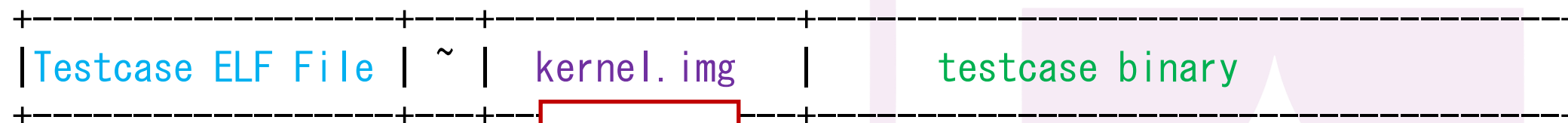
Physical Address

0x0

0x30000

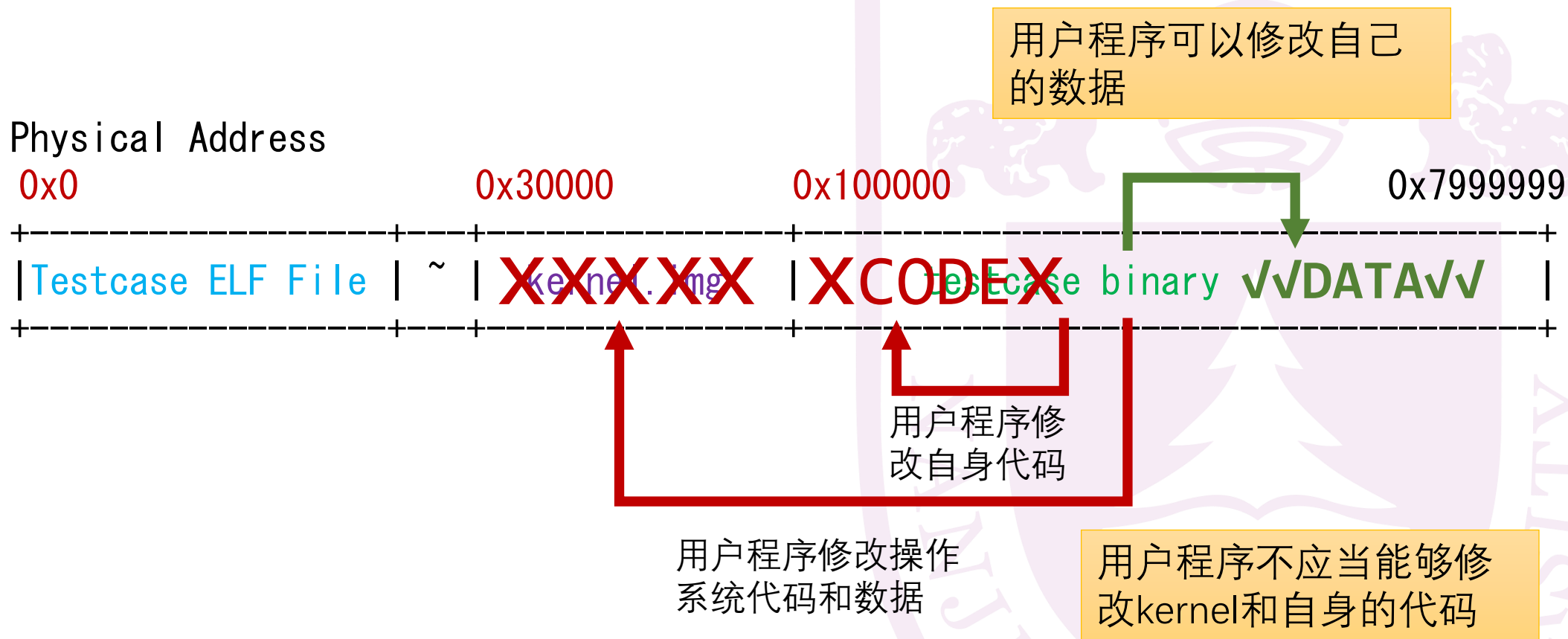
0x100000

0x7999999



loader()

kernel能够读写用户程序的内存区域,
是完成loader功能所必须的



具有不同功能的内存区域，应当为不同的程序规定不同的访问权限

分段机制的动机



把同一种功能和访问控制类型的数据归类放在一起，统一提供保护

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 1902

熟食只能厨师长才能存取
生食只要帮厨就能存取

.....

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 1902

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

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)

段表：我们指全局描述符表（GDT），LDT我们不模拟

段基址（base）	界限（limit）	权限要求（DPL）	其它
0x0	0x1000	0x0	...
0x1000	0xFFFF	0x3	...
...

段描述符

段描述符

低
地
址

高
地
址

段（segment）

段（segment）

段（segment）

0号段

1号段

0号段

1号段

段基址 (base)	界限 (limit)	权限要求 (DPL)	其它
0x0	0x1000	0x0	...
0x1000	0xFFFF	0x3	...
...

低
地
址

高
地
址

权限要求 (DPL)

段 (segment)	段 (segment)	段 (segment)
-------------	-------------	-------------

界限 (limit)

段基址 (base)

0号段

1号段

段基址 (base)	界限 (limit)	权限要求 (DPL)	其它
0x0	0x1000	0x0	...
0x1000	0xFFFF	0x3	...
...

低
地
址

高
地
址

权限要求 (DPL)

段 (segment)	段 (segment)	段 (segment)
-------------	-------------	-------------

界限 (limit)

段基址 (base)

程序访存时提供如下信息：

要访问内存的哪个段？

访问的数据相对段基址的偏移量是多少？

程序自身的权限等级是多少？

0号段

1号段

段基址 (base)	界限 (limit)	权限要求 (DPL)	其它
0x0	0x1000	0x0	...
0x1000	0xFFFF	0x3	...
...

低地址

高地址

权限要求 (DPL)

段 (segment)	段 (segment)	段 (segment)
-------------	-------------	-------------

界限 (limit)

段基址 (base)

确定段基址

程序访存时提供如下信息:

要访问内存的哪个段? **0号段**

访问的数据相对段基址的偏移量是多少?

程序自身的权限等级是多少?

0号段

1号段

段基址 (base)	界限 (limit)	权限要求 (DPL)	其它
0x0	0x1000	0x0	...
0x1000	0xFFFF	0x3	...
...

低地址

高地址

权限要求 (DPL)

段 (segment)	段 (segment)	段 (segment)
-------------	-------------	-------------

界限 (limit)

段基址 (base)

检查越界情况

程序访存时提供如下信息：

要访问内存的哪个段？ 0号段

访问的数据相对段基址的偏移量是多少？ 0x500

程序自身的权限等级是多少？

0号段

1号段

段基址 (base)	界限 (limit)	权限要求 (DPL)	其它
0x0	0x1000	0x0	...
0x1000	0xFFFF	0x3	...
...

低地址

高地址

权限要求 (DPL)



界限 (limit)

段基址 (base)

检查权限是否满足

程序访存时提供如下信息:

要访问内存的哪个段? **0号段**

访问的数据相对段基址的偏移量是多少? **0x500**

程序自身的权限等级是多少? **0x0**

低地址

高地址

权限要求 (DPL) : 高

权限要求 (DPL) : 低

代码段	数据段	其它段
-----	-----	-----

X

✓

可以正常访问

权限不足
segmentation fault

用户程序，权限等级低

低地址

高地址

权限要求 (DPL) : 高

权限要求 (DPL) : 低

代码段

数据段

其它段

界限 (limit)

段基址 (base)

假装访问
数据段

用户程序，权限等级低

给出偏移量超过界限
segmentation fault

1902

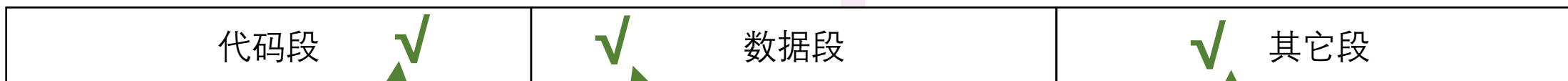
UNIVERSITY

低地址

高地址

权限要求 (DPL) : 高

权限要求 (DPL) : 低



可以实现
代码和数
据的装载

操作系统，权限等级高

段表：我们指全局描述符表（GDT），LDT我们不模拟

段基址（base）	界限（limit）	权限要求（DPL）	其它
0x0	0x1000	0x0	...
0x1000	0xFFFF	0x3	...
...

段描述符

段描述符

分段机制的运行需要计算机和OS相配合
GDT由OS初始化，保存在内存中
GDTR寄存器保存GDT的首地址和界限（GDT长度-1）
开启分段机制后的地址转换和保护检查由机器完成

低
地
址

高
地
址

段（segment）	段（segment）	段（segment）
------------	------------	------------



分段机制的具体实现方案

分段机制：实模式 – 现在NEMU的访存模式

`movl 0x12345678, %eax // 指令给出地址`

`operand_read()`

`vaddr = 0x12345678`

`vaddr_read(vaddr_t vaddr, ...)`

`laddr = vaddr`

`laddr_read(laddr_t laddr, ...)`

`paddr = laddr`

`paddr_read(paddr_t paddr, ...)`

`paddr = paddr`

`cache_read(paddr_t paddr, ...)`

`paddr = paddr`

`hw_mem_read(paddr_t paddr, ...)`

`hw_mem[paddr]`

`paddr = 0x12345678`

- 指令给出的地址即为物理地址
- 中间没有经过任何的权限检查

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：实模式 – 现在NEMU的访存模式

- 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：启动过程

- 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：启动之后

- 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

程序访存时提供如下信息：

要访问内存的哪个段？ **0号段**

访问的数据相对段基址的偏移量是多少？ **0x500**

程序自身的权限等级是多少？ **0x0**

分段机制

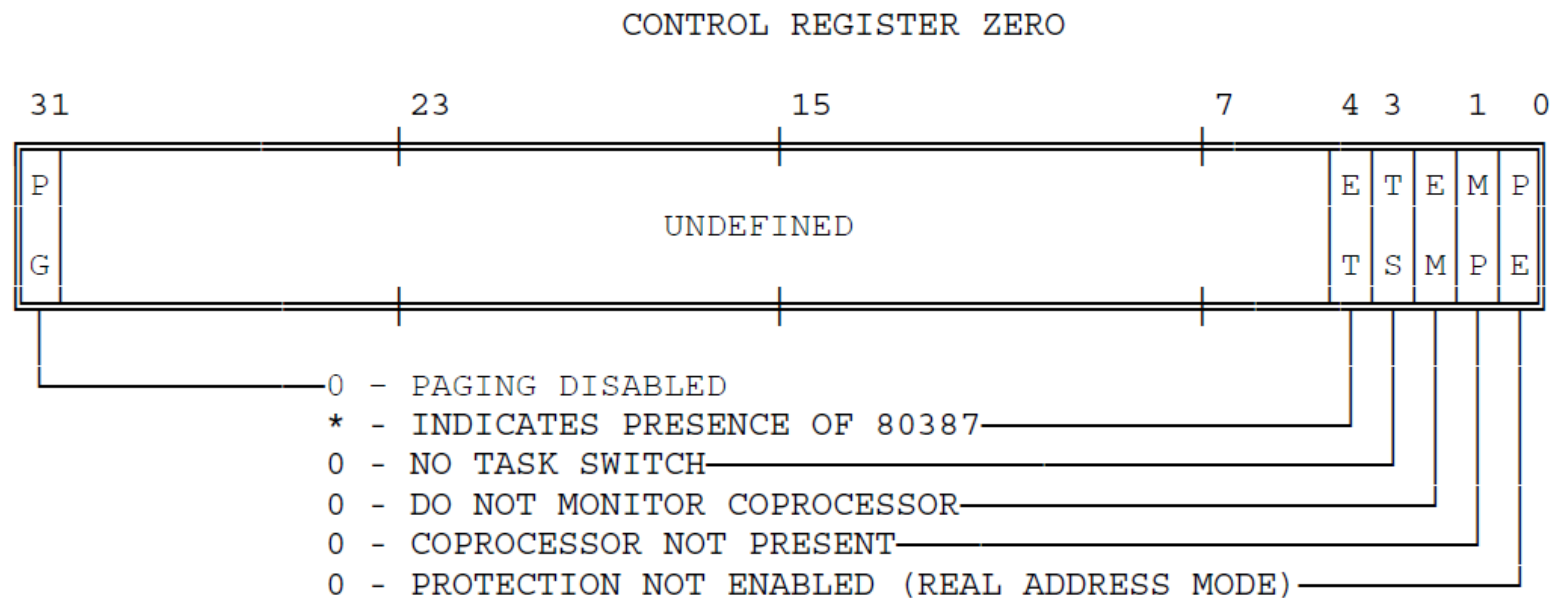
- x86的机器开机后首先进入实模式
 - 加载操作系统
 - 操作系统初始化段表
- ③ 在代码阶段讲
- ① • 拨动一个‘开关’，从实模式切换到保护模式（开启分段机制）
 - 进入保护模式后
- ② • 程序给出48位逻辑地址（16位段选择符 + 32位有效地址）
 - 使用段选择符来查段表
 - 进行段级地址转换得到线性（现在就是物理）地址
- PPT的讲述次序，注意和开机后的执行次序不同

分段机制（那个‘开关’）

在NEMU中，CR0寄存器如何实现？

- 80386的实模式和保护模式
 - 由CR0寄存器中的PE位控制

1. 参考EFLAGS寄存器（注意大小端序）
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

- 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

2

程序访存时提供如下信息：

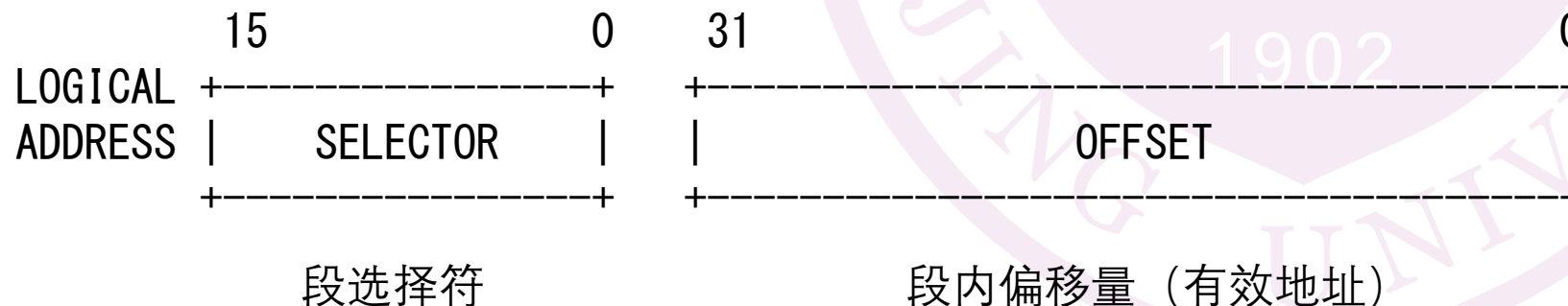
要访问内存的哪个段？ **0号段**

访问的数据相对段基址的偏移量是多少？ **0x500**

程序自身的权限等级是多少？ **0x0**

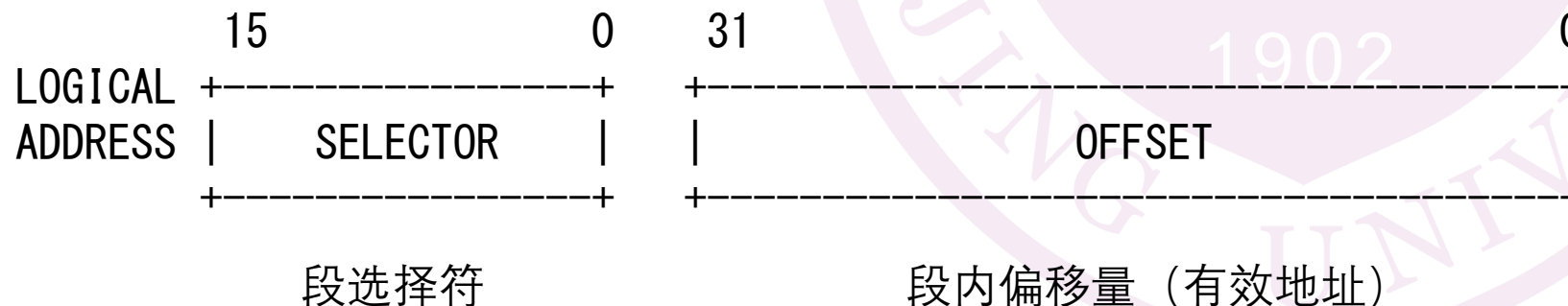
分段机制（地址转换）

- 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

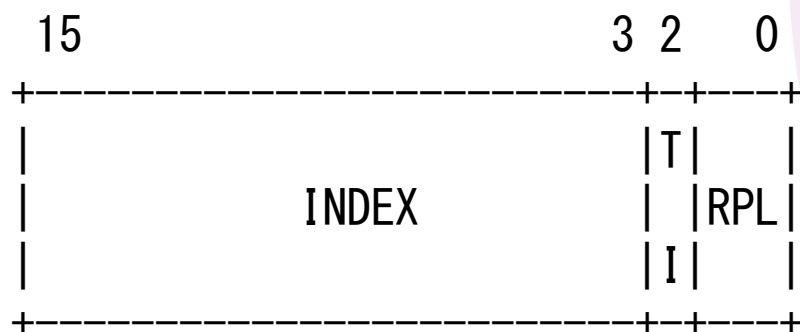


分段机制 (地址转换)

- 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

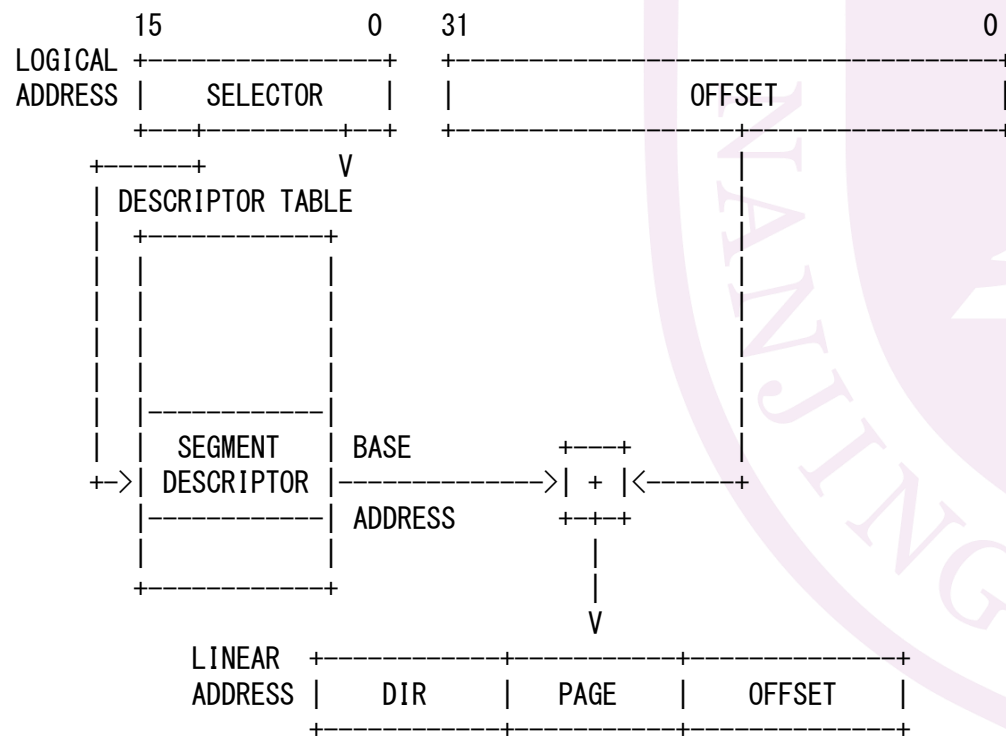
11表示最低级，用户态

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

段基址 (base)	界限 (limit)	权限要求 (DPL)	其它
0x0	0x1000	0x0	...
0x1000	0xFFFF	0x3	...
...

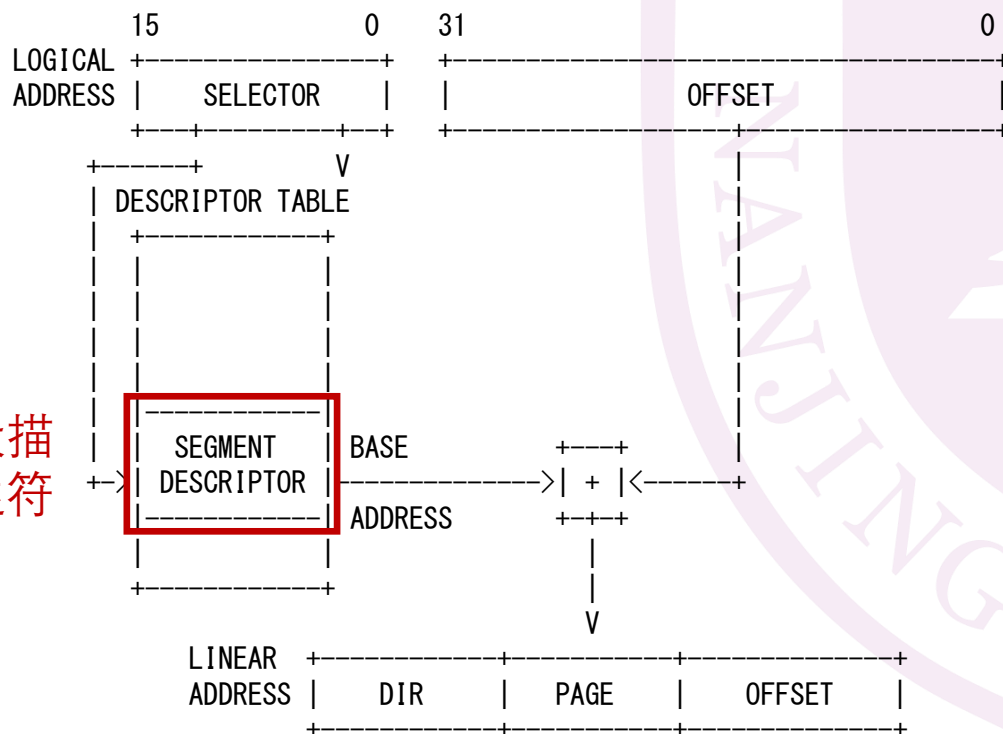


分段机制（地址转换）

- 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

段基址 (base)	界限 (limit)	权限要求 (DPL)	其它
0x0	0x1000	0x0	...
0x1000	0xFFFF	0x3	...
...

段描述符



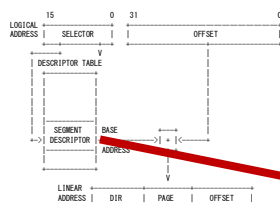
分段机制（地址转换）

段表：我们指全局描述符表（GDT），LDT我们不模拟

段基址 (base)	界限 (limit)	权限要求 (DPL)	其它
0x0	0x1000	0x0	...
0x1000	0xFFFF	0x3	...
...

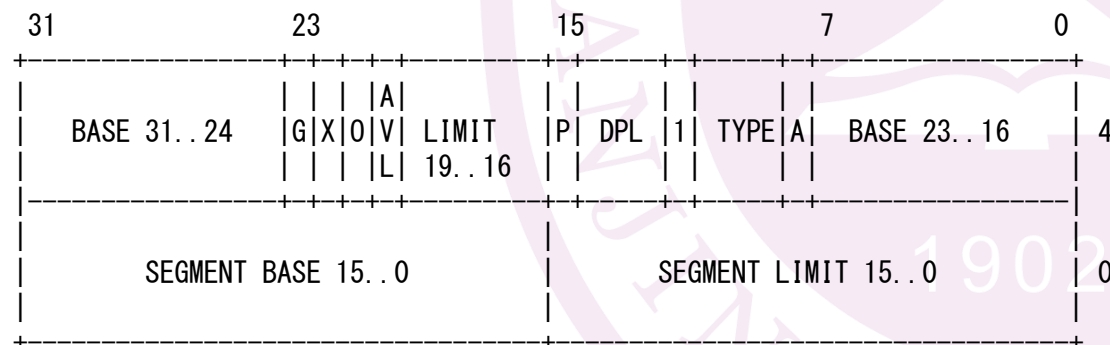
段描述符
段描述符

- 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

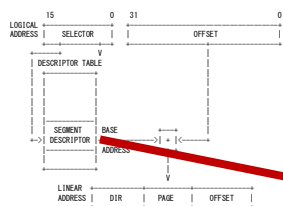
DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS



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

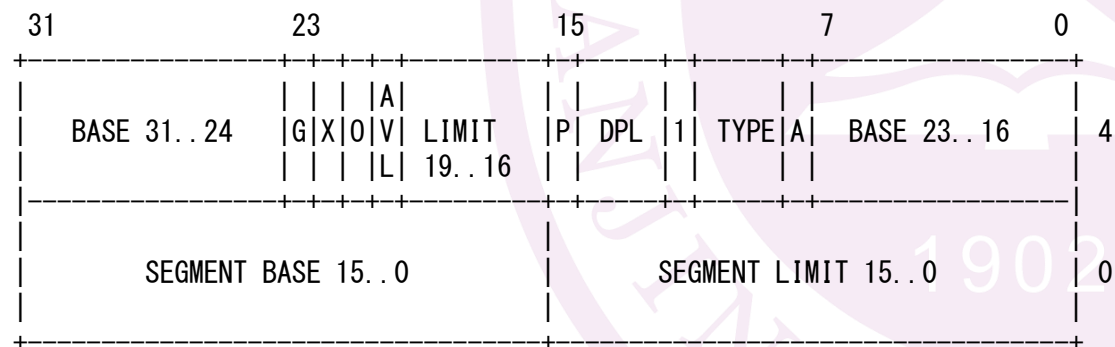
分段机制（地址转换）

- 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

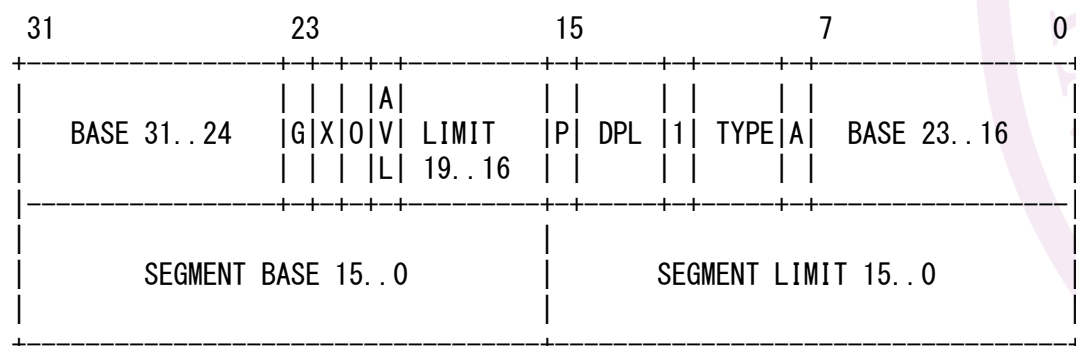


A - ACCESSED
AVL - AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
DPL - DESCRIPTOR PRIVILEGE LEVEL
G - GRANULARITY
P - SEGMENT PRESENT

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



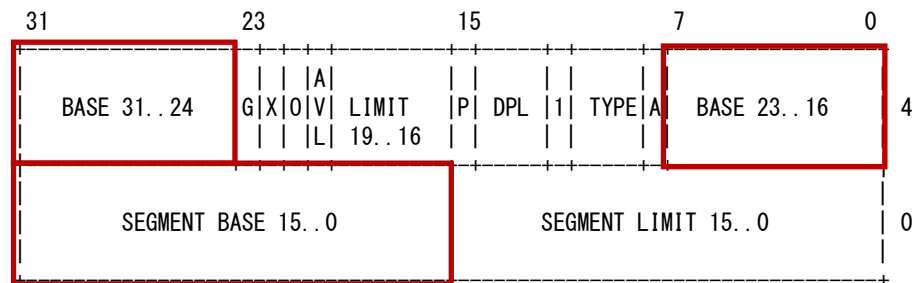
A - ACCESSED
AVL - AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
DPL - DESCRIPTOR PRIVILEGE LEVEL
G - GRANULARITY
P - SEGMENT PRESENT

```
/* 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`

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;
```

32位基地址

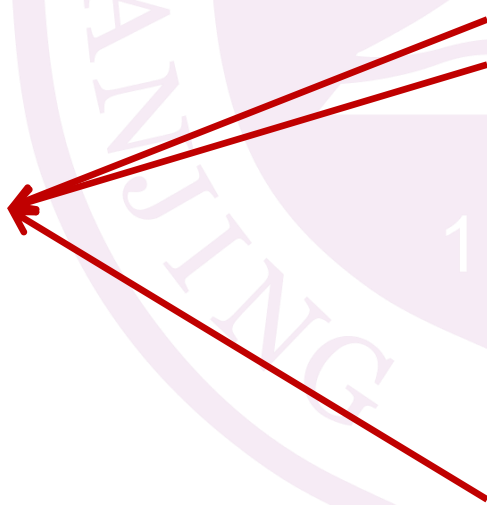
分段机制（地址转换）

- 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`

32位线性地址 =
32位基地址
+ 32位段内偏移量（有效地址）

32位基地址

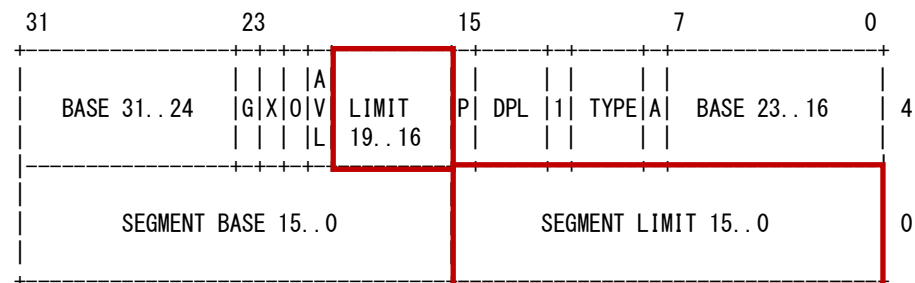
```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 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`

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

20位界限

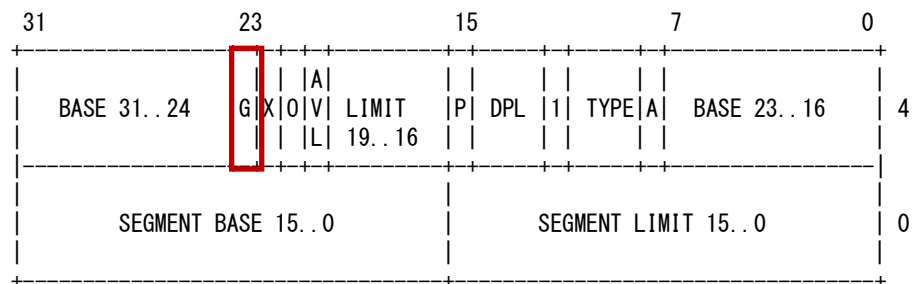
指出段的长度，用于检查地址越界，及偏移量超出最大段长的情况

```
/* 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`

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;
```

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

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`

配合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`

配合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？ $2^{20} \times 4\text{KB} = 4\text{GB}$

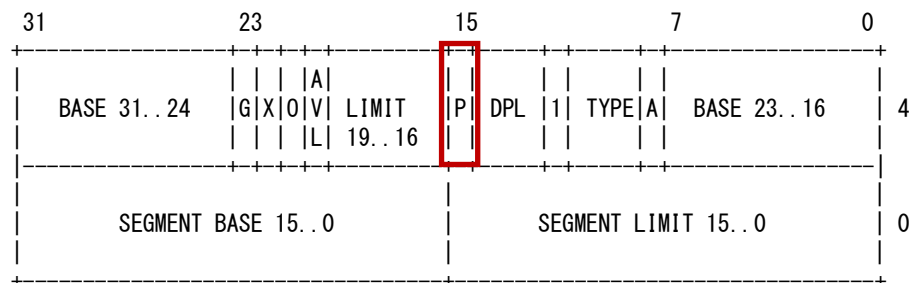
粒度大小：为1表示段以页（4KB）为基本单位，为0表示以字节为基本单位

```
/* 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`

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

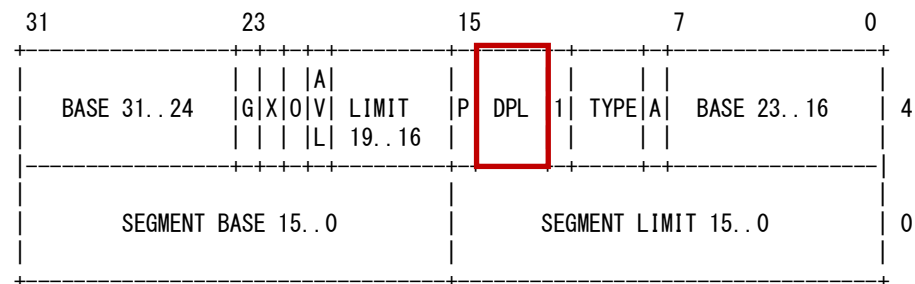
```
/* the 64bit segment descriptor */
typedef struct SegmentDescriptor {
    uint32_t limit_15_0      : 16;
    uint32_t base_15_0      : 16;
    uint32_t base_23_16     : 8;
    uint32_t type            : 4;
    uint32_t segment_type    : 1;
    uint32_t privilege_level : 2;
    uint32_t present         : 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;
```

存在位：为1表示段
已在内存中，为0表
示段不在内存中

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



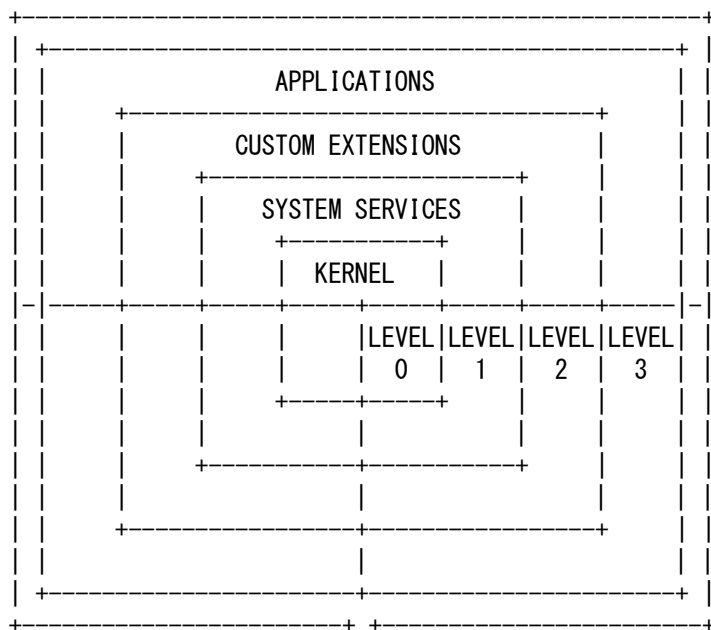
A - ACCESSED
AVL - AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
DPL - DESCRIPTOR PRIVILEGE LEVEL
G - GRANULARITY
P - SEGMENT PRESENT

```
/* 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;
```

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`



rings

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`

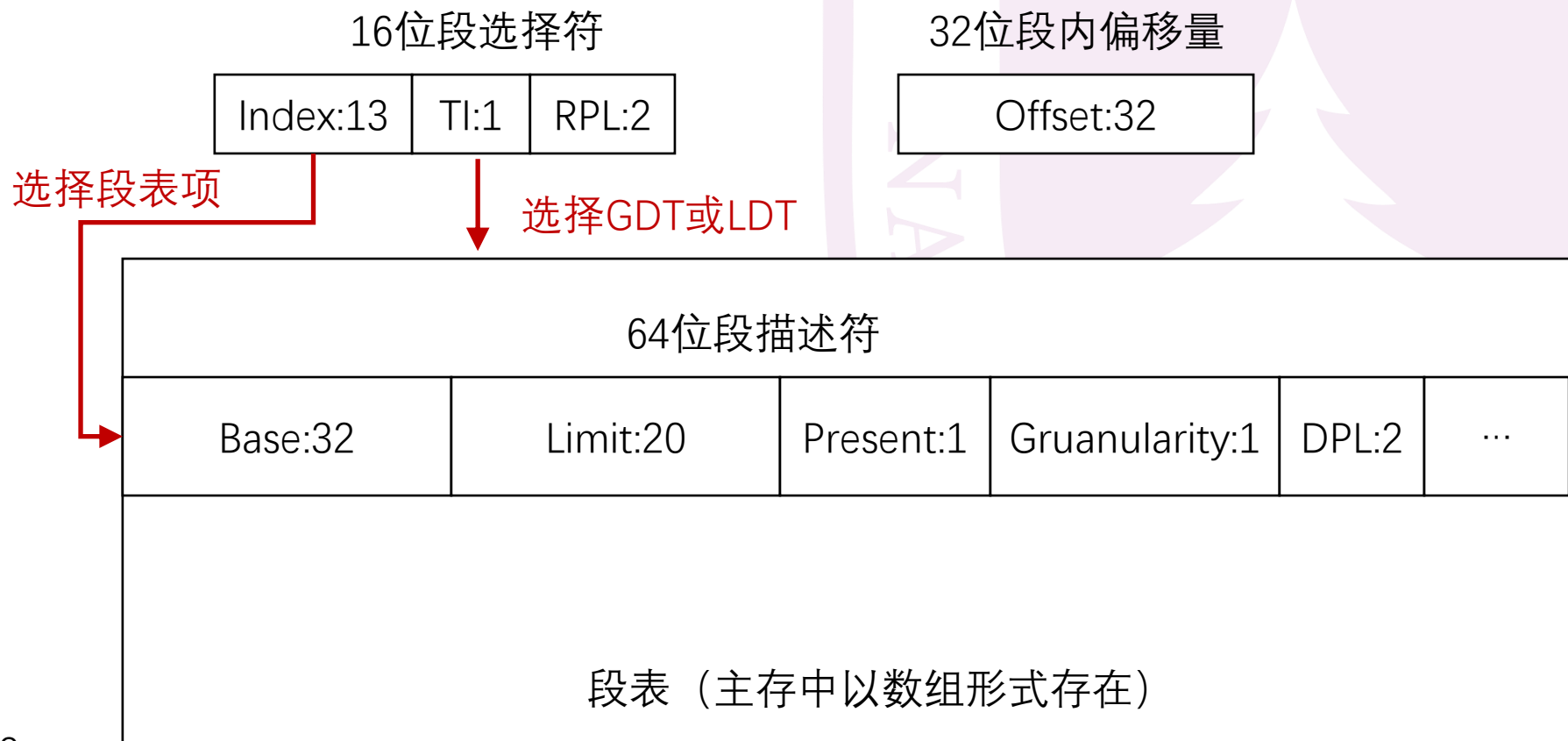
只有当从数值上：
段描述符的DPL \geq 段选择符的RPL
段描述符的DPL \geq 进程的CPL
才有权访问该段

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	Tl: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	Tl: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	Tl:1	RPL:2
----------	------	-------

32位段内偏移量

Offset:32

64位段描述符

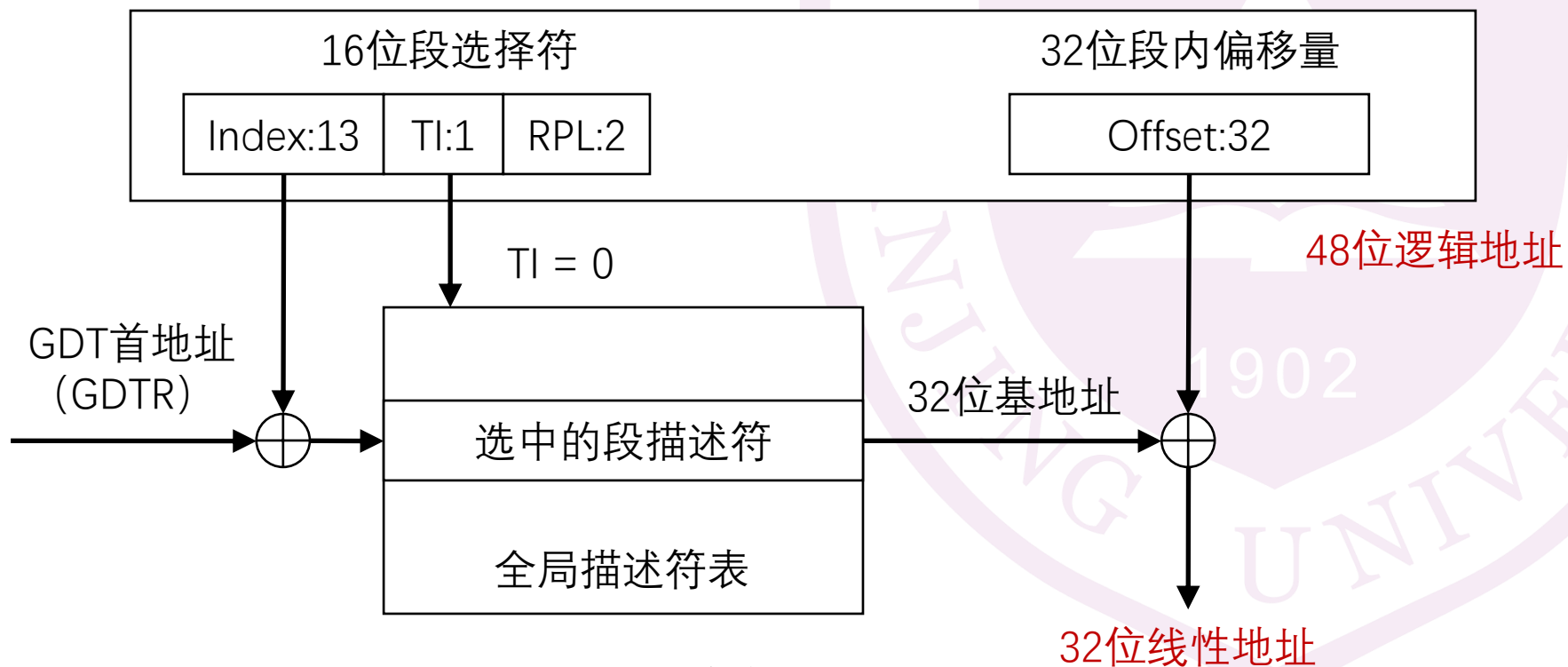
Base:32	Limit:20	Present:1	Gruanularity:1	DPL:2	...
---------	----------	-----------	----------------	-------	-----

线性地址=Base+Offset

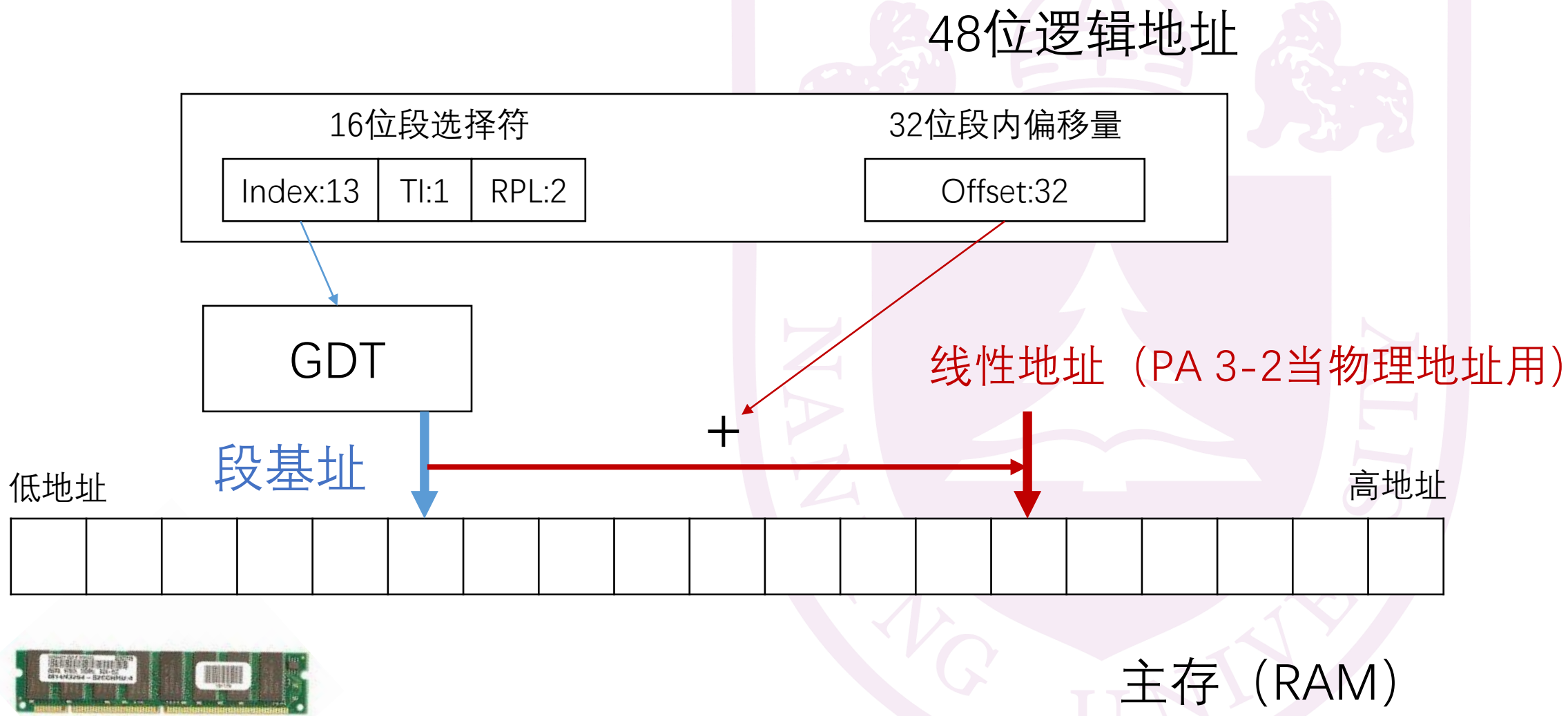
段表（主存中以数组形式存在）

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）



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



PA中对于分段机制的模拟

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(gtdtdesc)
# 这里在干啥?
    ljmp    $GDT_ENTRY(1), $va_to_pa(start_cond) #这里在干啥?
start_cond:
# Set up the protected-mode data segment registers
    movw    $GDT_ENTRY(2), %ax
    movw    %ax, %ds          # %DS = %AX
    ...
# Enable protection
    movl    %cr0, %eax        # %CR0 |= PROTECT_ENABLE_BIT
    orl     $0x1, %eax
    movl    %eax, %cr0
# Set up a stack for C code.
    ...          jmp init    # never return
```

kernel/start/start.S

下一页

分段机制 (Kernel初始化段表)

接上一页

```
#define MAKE_SEG_DESC(type, base, lim) \ ...  
  
# GDT  
.p2align 2  
gdt:  
    MAKE_NULL_SEG_DESC                # empty segment  
    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1 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

[nemu/src/memory/memory.c](#)


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

分段机制（勘误）

Two-Byte Opcode Map (first byte is 0FH)

	0	1	2	3
0	Grp6	Grp7	LAR Gw, Ew	LSL Gv, Ew
1				
2	MOV Cd, Rd	MOV Dd, Rd	MOV Rd, Cd	MOV Rd, Dd



总结

- Include/config.h中定义#define IA32_SEG
- 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 3-1截止时间
2020年12月16日24时



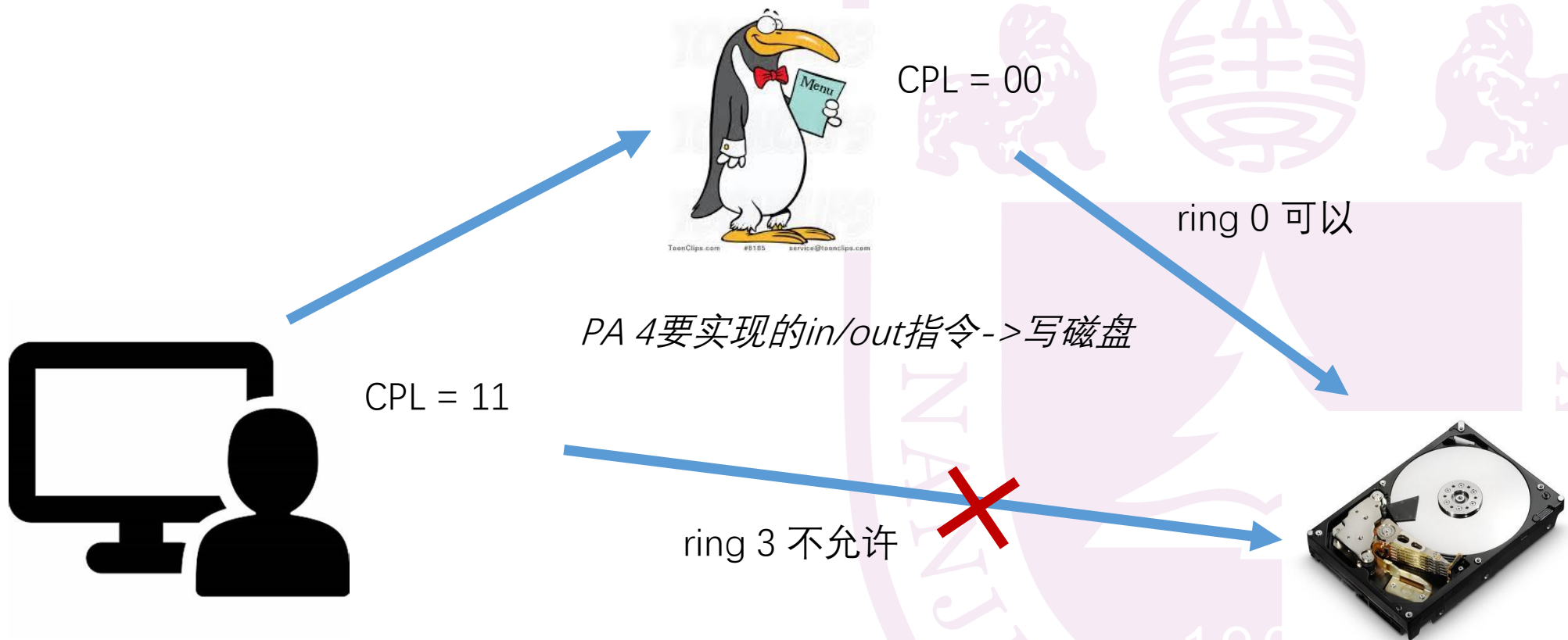
PA 3-2截止时间
2020年12月23日24时

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

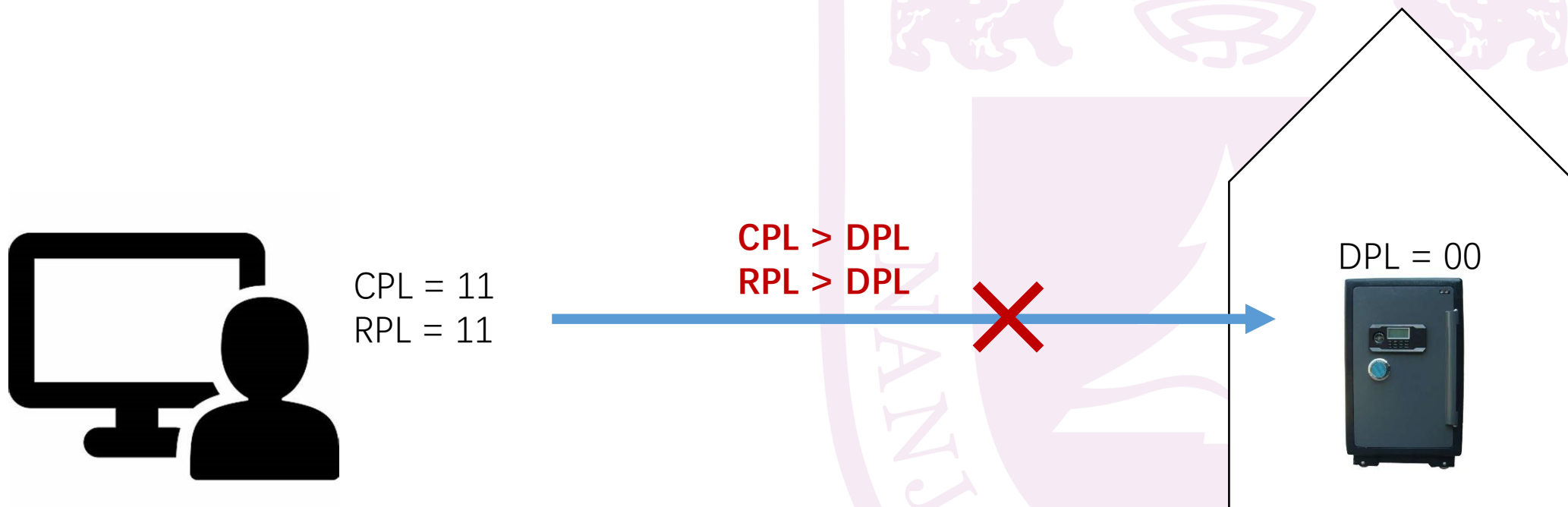
特权等级为什么要涉及CPL和RPL两个东西



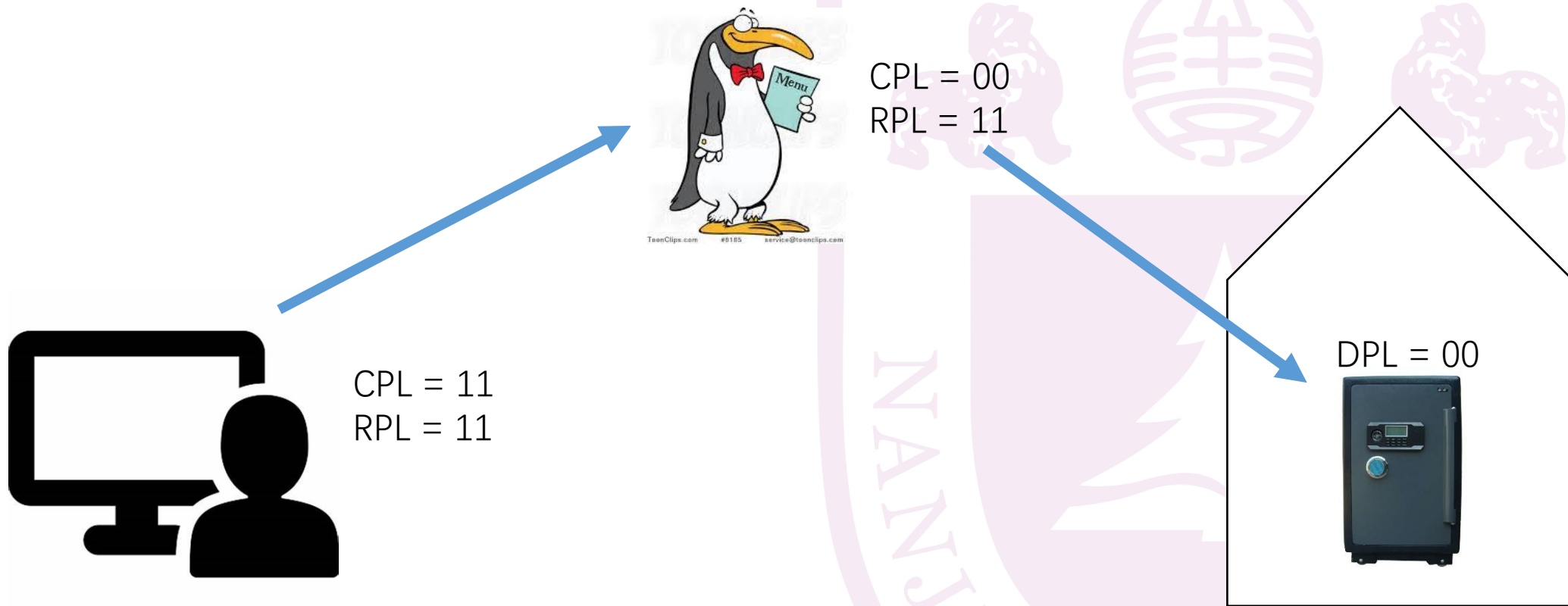
特权等级为什么要涉及CPL和RPL两个东西



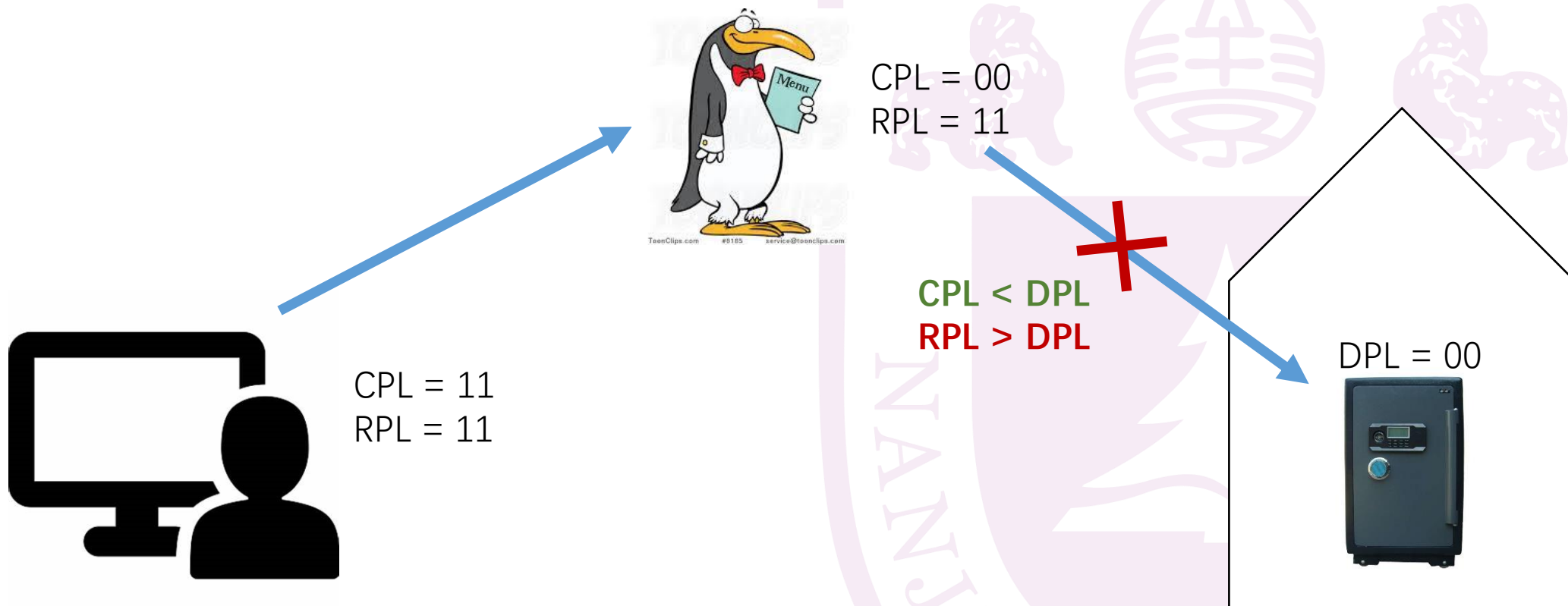
特权等级为什么要涉及CPL和RPL两个东西



特权等级为什么要涉及CPL和RPL两个东西



特权等级为什么要涉及CPL和RPL两个东西



The RPL can be used to insure that privileged code does not access a segment on behalf of an application program unless the program itself has access privileges for that segment.