第二章　内存寻址

前言
程序中的代码和变量（数据）

- 代码 → 子程序中语句

- 变量（数据） → 程序中定义的变量
  - 局部变量 定义在子程序内的变量
  - 全局变量 定义在所有子程序外部的变量
    - 带初值的 data
    - 不带初值的 bss
  - 只读变量 常数字符串……

```
#include <stdio.h>
#include <stdlib.h>
int matrix[3][3];

int main()
{
   int i,j;
   int count ;
   FILE *out;

line1:    out = fopen("Matrix","w");
line2:    for( i=0; i<3; i++)
line3:       for( j=0; j<3; j++)
line4:          matrix[i][j]) = count++;

line5:    square(3,3);

line6:    for( i=0; i<3; i++)
line7:    {
line8:       for( j=0; j<3; j++)
line9:
          fprintf(out,"%d\t",matrix[i][j]
);
line10:             fprintf(out,"\n");
line11:   }
line12:   printf("Program Finished!\n");
line13:   exit(0);
}

void square(int m, int n)
{
          int row,line;
line14:   for( row=0; row<m; row++)
line15:      for( line=0; line<n; line++)
line16:         matrix[row][line] *=
                              matrix[row][line];
}
```

前言
静态链接程序中的段

- **1个**代码段（text）：　　子程序集合
- 其余段：　　变量（数据）集合
  - 局部变量　　**1个**堆栈段 (stack)
  - 全局变量
    - 带初值的　　**1个**数据段 (data)
    - 不带初值的　**1个**BSS（bss）
  - 只读变量　　**1个**只读数据段 (ronly)

```
#include <stdio.h>
#include <stdlib.h>
int matrix[3][3];

int main()
{
   int i,j;
   int count ;
   FILE *out;

line1:    out = fopen("Matrix","w");
line2:    for( i=0; i<3; i++)
line3:       for( j=0; j<3; j++)
line4:          matrix[i][j]) = count++;

line5:    square(3,3);

line6:    for( i=0; i<3; i++)
line7:    {
line8:       for( j=0; j<3; j++)
line9:
          fprintf(out,"%d\t",matrix[i][j]
);
line10:             fprintf(out,"\n");
line11:   }
line12:   printf("Program Finished!\n");
line13:   exit(0);
}

void square(int m, int n)
{
          int row,line;
line14:   for( row=0; row<m; row++)
line15:      for( line=0; line<n; line++)
line16:         matrix[row][line] *=
                              matrix[row][line];
}
```

前言

进程（执行静态链接的程序）中的段

- **1个**代码段（text）：　　子程序集合
- 其余段：　　变量（数据）集合
  - 局部变量　　**1个**堆栈段 (stack)
  - 全局变量
    - 带初值的　　1个数据段 (data)
    - 不带初值的　1个BSS（bss）
  - 只读变量　　1个只读数据段 (ronly)

1个数据段

执行同一个应用程序的多个进程，
共享内存中的一份代码。
数据段、堆栈段不共享。每个进程用自己的。

进程运行起来之后，还会生成其它的段嘛？

会的。一个heap、许多mmap的段

前言
动态链接使用的段

- 共享库（动态链接库）中的段
  - 每个库3个段：代码段、数据段、BSS段
- 进程中，还包括
  - ld程序的代码段、数据段和BSS段
  - 执行应用程序前，进程会执行ld程序，确定 共享库中函数和变量 的地址
  - ld程序执行完毕后，所有指令和变量地址得到确定，进程 exec 执行应用程序

```
#include <stdio.h>
#include <stdlib.h>
int matrix[3][3];

int main()
{
   int i,j;
   int count ;
   FILE *out;

line1:    out = fopen("Matrix","w");
line2:    for( i=0; i<3; i++)
line3:       for( j=0; j<3; j++)
line4:          matrix[i][j]) = count++;

line5:    square(3,3);

line6:    for( i=0; i<3; i++)
line7:    {
line8:       for( j=0; j<3; j++)
line9:
          fprintf(out,"%d\t",matrix[i][j]
);
line10:             fprintf(out,"\n");
line11:   }
```

```
line12:   printf("Program Finished!\n");
line13:   exit(0);
}

void square(int m, int n)
{
          int row,line;
line14:   for( row=0; row<m; row++)
line15:      for( line=0; line<n; line++)
line16:         matrix[row][line] *=
                              matrix[row][line];
}
```

前言

进程（执行动态链接的程序）中的段

- **1个**代码段（text）
- **1个**数据段（data）

} 应用程序的内容

- 每include一个共享库
  - **1个**代码段（text）
  - **1个**数据段（data）
  - 1个 0 页 （bss）

+ ld 程序的3个段

- 1个栈 所有子程序的栈帧
- 1个heap 如果进程有malloc（new）
- 好多mmap段 每mmap一个磁盘文件就会有一个mmap段

前言
私有段 和 共享段

- **1个**代码段（text）
- **1个**数据段（data）

- 每include一个共享库
  - **1个**代码段（text）
  - **1个**数据段（data）
  - 1个 0 页　（bss）

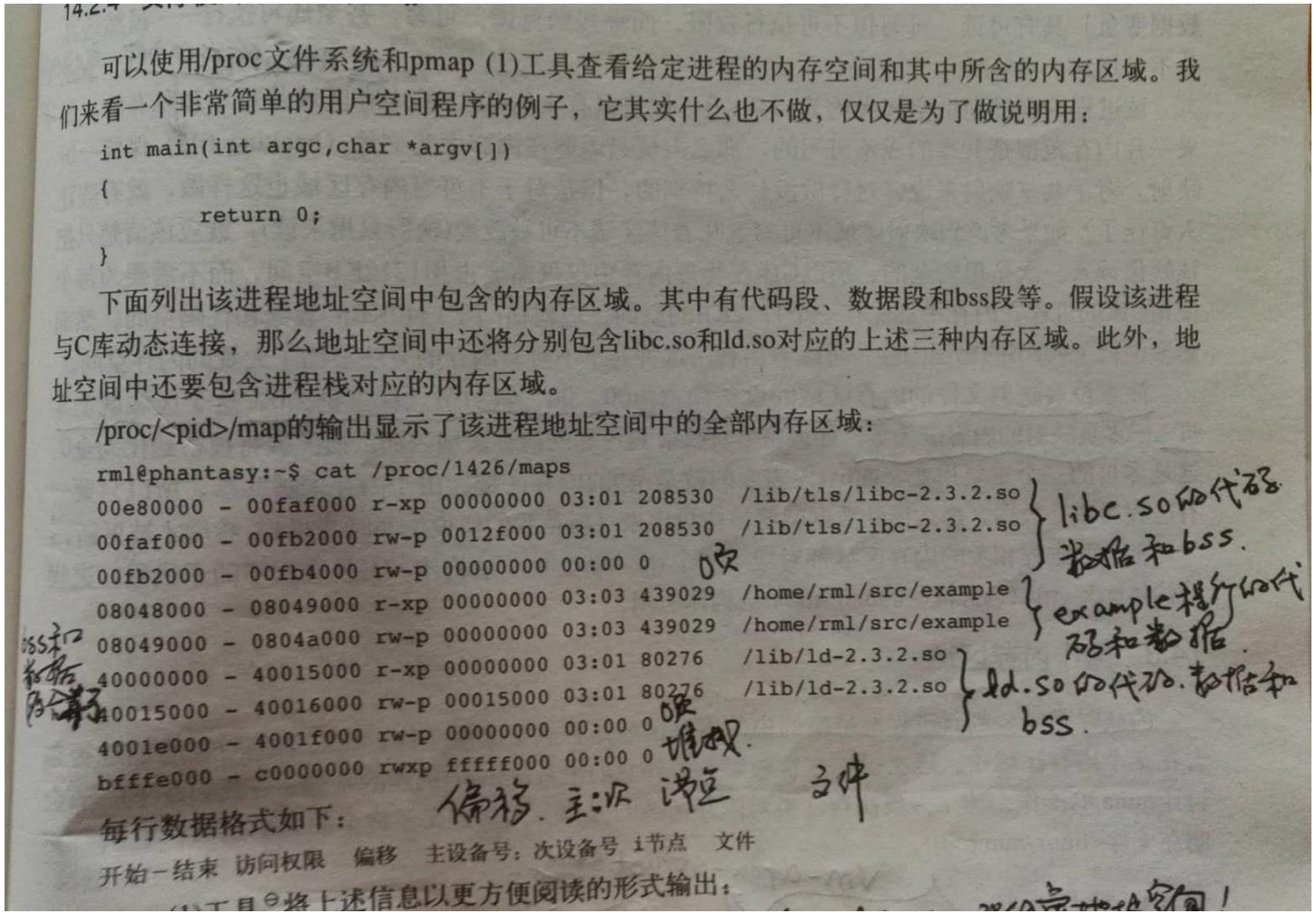
- 1个栈　　所有子程序的栈帧

- 1个heap　　如果进程有malloc（new）

- 好多mmap段　　每mmap一个磁盘文件就会有一个mmap段

Linux进程的虚地址空间布局（进程内存布局）

例



可以使用/proc文件系统和pmap (1)工具查看给定进程的内存空间和其中所含的内存区域。我们来看一个非常简单的用户空间程序的例子，它其实什么也不做，仅仅是为了做说明用：

```
int main(int argc,char *argv[])
{
        return 0;
}
```

下面列出该进程地址空间中包含的内存区域。其中有代码段、数据段和bss段等。假设该进程与C库动态连接，那么地址空间中还将分别包含libc.so和ld.so对应的上述三种内存区域。此外，地址空间中还要包含进程栈对应的内存区域。

/proc/<pid>/map的输出显示了该进程地址空间中的全部内存区域：

```
rml@phantasy:~$ cat /proc/1426/maps
00e80000 - 00faf000 r-xp 00000000 03:01 208530  /lib/tls/libc-2.3.2.so
00faf000 - 00fb2000 rw-p 0012f000 03:01 208530  /lib/tls/libc-2.3.2.so
00fb2000 - 00fb4000 rw-p 00000000 00:00 0
08048000 - 08049000 r-xp 00000000 03:03 439029  /home/rml/src/example
08049000 - 0804a000 rw-p 00000000 03:03 439029  /home/rml/src/example
40000000 - 40015000 r-xp 00000000 03:01 80276   /lib/ld-2.3.2.so
40015000 - 40016000 rw-p 00015000 03:01 80276   /lib/ld-2.3.2.so
4001e000 - 4001f000 rw-p 00000000 00:00 0
bfffe000 - c0000000 rwxp fffff000 00:00 0
```

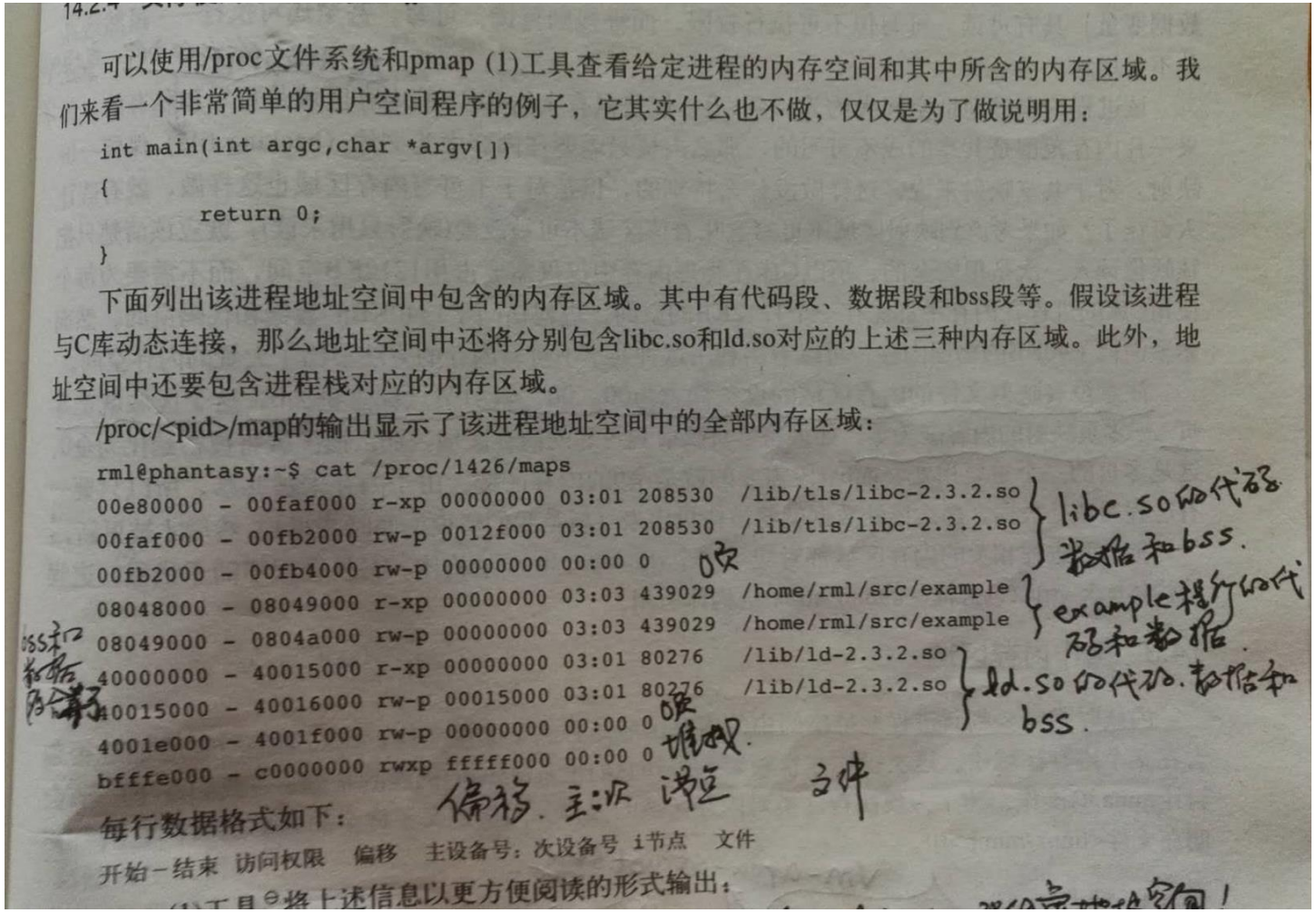
每行数据格式如下：

开始－结束 访问权限 偏移 主设备号：次设备号 i节点 文件

（1）工具⊖将上述信息以更方便阅读的形式输出：

作业：
观察进程的虚地址空间布局

helloWorld程序



可以使用/proc文件系统和pmap (1)工具查看给定进程的内存空间和其中所含的内存区域。我们来看一个非常简单的用户空间程序的例子，它其实什么也不做，仅仅是为了做说明用：

```
int main(int argc,char *argv[])
{
        return 0;
}
```

下面列出该进程地址空间中包含的内存区域。其中有代码段、数据段和bss段等。假设该进程与C库动态连接，那么地址空间中还将分别包含libc.so和ld.so对应的上述三种内存区域。此外，地址空间中还要包含进程栈对应的内存区域。

/proc/<pid>/map的输出显示了该进程地址空间中的全部内存区域：

```
rml@phantasy:~$ cat /proc/1426/maps
00e80000 - 00faf000 r-xp 00000000 03:01 208530  /lib/tls/libc-2.3.2.so
00faf000 - 00fb2000 rw-p 0012f000 03:01 208530  /lib/tls/libc-2.3.2.so
00fb2000 - 00fb4000 rw-p 00000000 00:00 0
08048000 - 08049000 r-xp 00000000 03:03 439029  /home/rml/src/example
08049000 - 0804a000 rw-p 00000000 03:03 439029  /home/rml/src/example
40000000 - 40015000 r-xp 00000000 03:01 80276   /lib/ld-2.3.2.so
40015000 - 40016000 rw-p 00015000 03:01 80276   /lib/ld-2.3.2.so
4001e000 - 4001f000 rw-p 00000000 00:00 0
bfffe000 - c0000000 rwxp fffff000 00:00 0
```

每行数据格式如下：

开始－结束 访问权限 偏移 主设备号：次设备号 i节点 文件

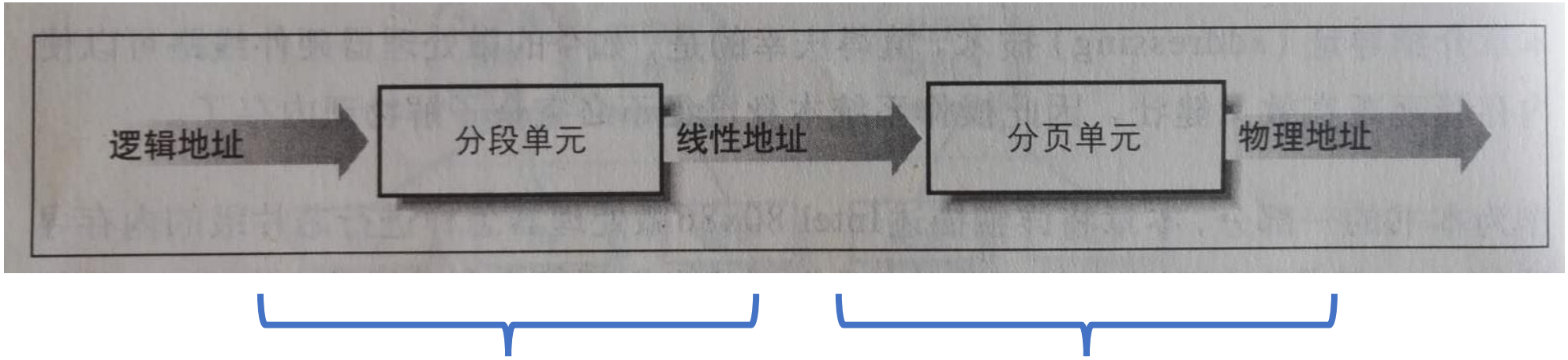
将上述信息以更方便阅读的形式输出：

第二章　内存寻址

Intel 80X86

3 种不同的地址

- 逻辑地址
- 线性地址
- 物理地址



段级映射：程序中的逻辑段落 到
进程虚地址（线性地址）空间的映射

页级映射：虚地址空间到物理内存的映射

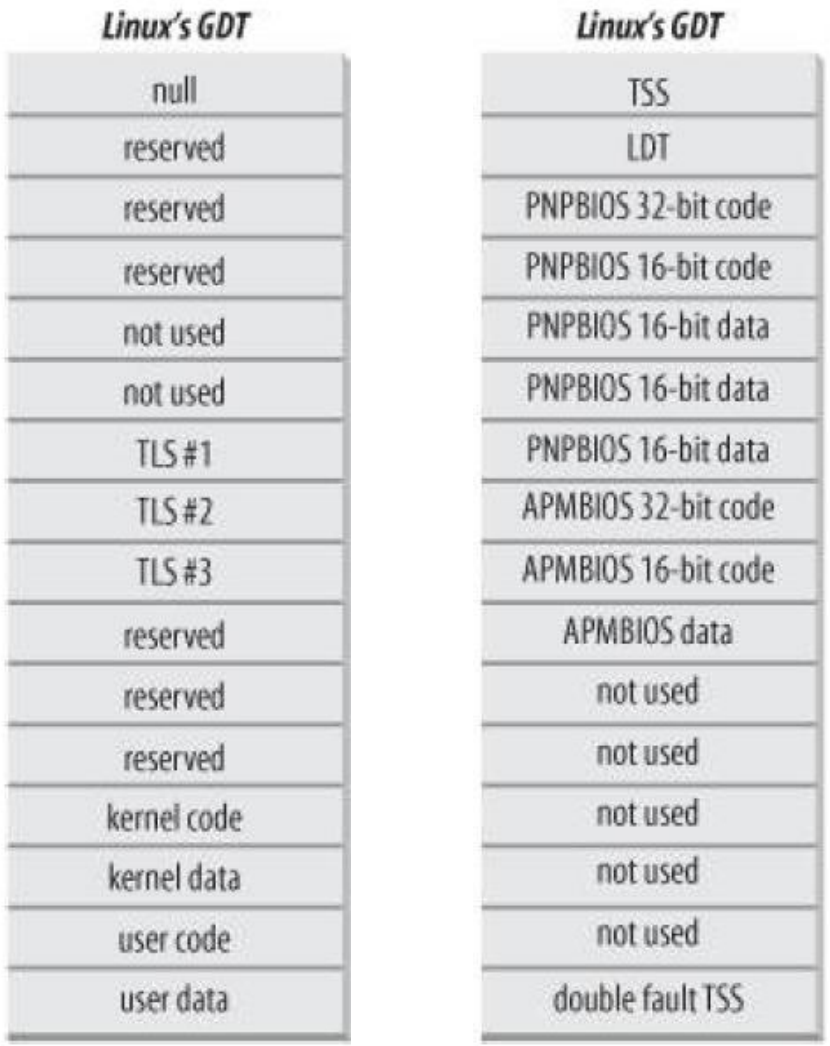
逻辑地址 和 线性地址

- 以CPU要取的下条指令为例

- 逻辑地址： CS：EIP。 解读：
  - CS 引用一个代码段。进程想要访问的那条指令在这个段里。
  - EIP 是段内偏移量offset。是 进程想要访问的那条指令，在段中的起始地址。

- 线性地址： CS引用的段的起始地址 + offset。 解读：
  - 这条指令在虚空间中的起始地址。

逻辑地址 → 线性地址　　GDT　P60



GDT是段描述符数组

内核代码段　GDT[12]
内核数据段　GDT[13]
用户代码段　GDT[14]
用户数据段　GDT[15]

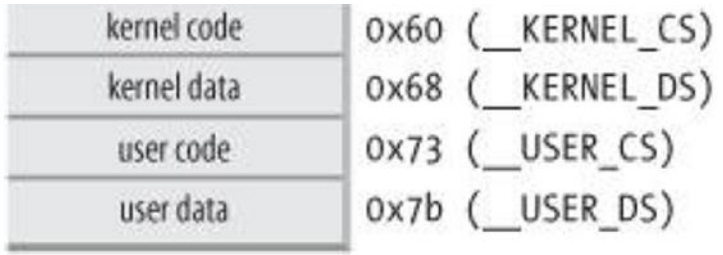
表 2-3：四个主要的 Linux 段的段描述符字段的值

段	Base	G	Limit	S	Type	DPL	D/B	P
用户代码段	0x00000000	1	0xfffff	1	10	3	1	1
用户数据段	0x00000000	1	0xfffff	1	2	3	1	1
内核代码段	0x00000000	1	0xfffff	1	10	0	1	1
内核数据段	0x00000000	1	0xfffff	1	2	0	1	1

以4K字节为单位

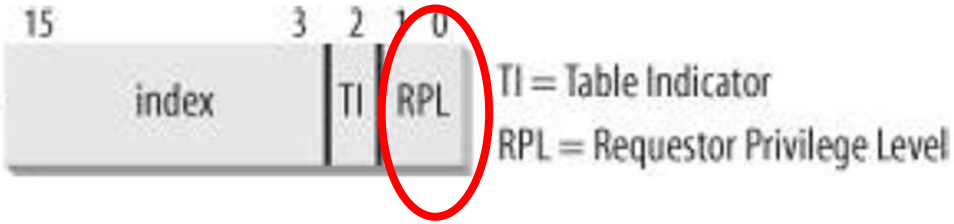
逻辑地址 → 线性地址　　段选择子

内核代码段　GDT[12]
内核数据段　GDT[13]
用户代码段　GDT[14]
用户数据段　GDT[15]



CPL（Current Privilege Level，CPU当前特权级）

CS



Linux的分段（i386架构）

CPU

GDTR

CS

CS引用的GDT表项的CPU复本

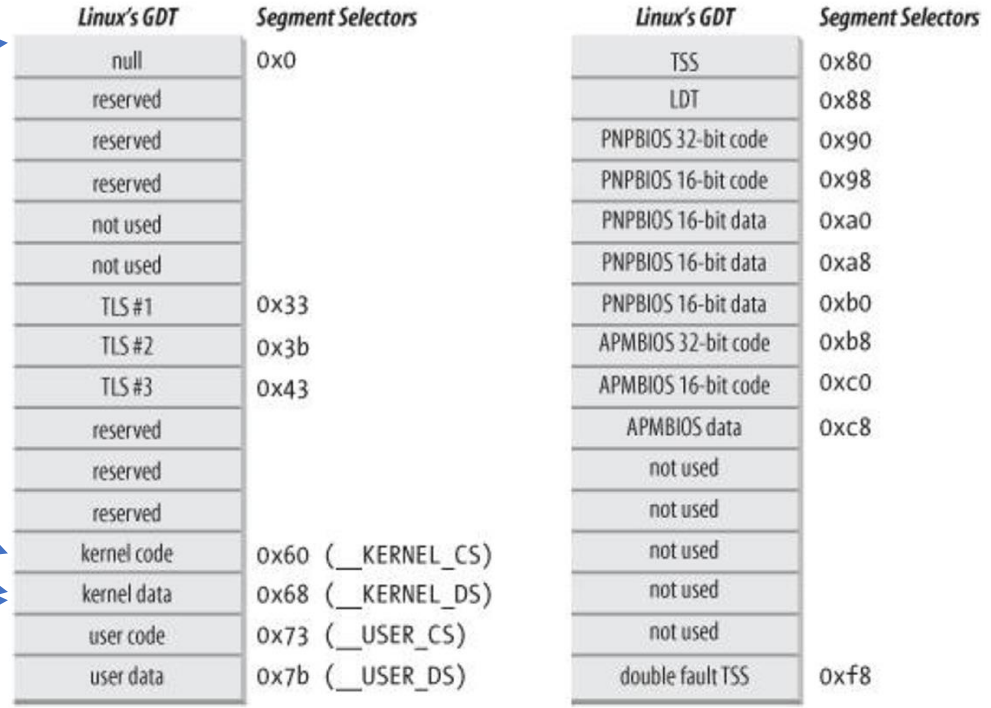
DS

DS引用的GDT表项的CPU复本

SS

SS引用的GDT表项的CPU复本

内存中的GDT



Linux's GDT	Segment Selectors
null	0x0
reserved	
reserved	
reserved	
not used	
not used	
TLS #1	0x33
TLS #2	0x3b
TLS #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)

Linux's GDT	Segment Selectors
TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
not used	
not used	
not used	
not used	
double fault TSS	0xf8

… 另外的3个段寄存器 …

P: 60

内核代码段　GDT[12]
内核数据段　GDT[13]
用户代码段　GDT[14]
用户数据段　GDT[15]

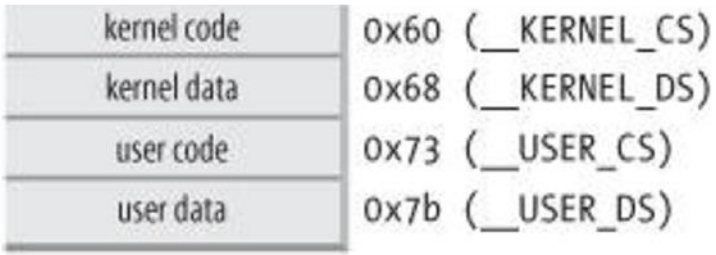


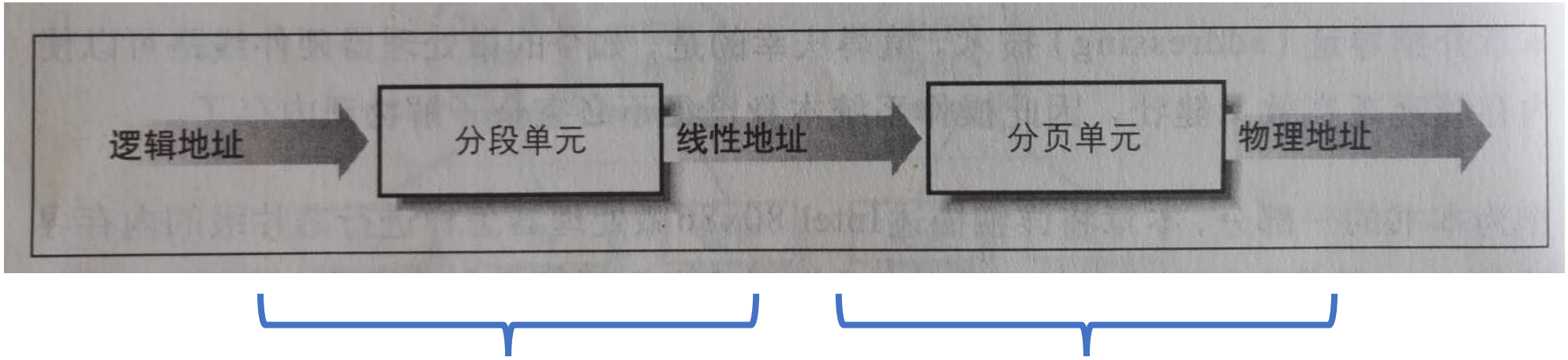
表 2-3：四个主要的 Linux 段的段描述符字段的值

段	Base	G	Limit	S	Type	DPL	D/B	P
用户代码段	0x00000000	1	0xfffff	1	10	3	1	1
用户数据段	0x00000000	1	0xfffff	1	2	3	1	1
内核代码段	0x00000000	1	0xfffff	1	10	0	1	1
内核数据段	0x00000000	1	0xfffff	1	2	0	1	1

以4K字节为单位

3 种不同的地址

- 逻辑地址
- 线性地址
- 物理地址



段级映射：程序中的逻辑段落 到
进程虚地址（线性地址）空间的映射

页级映射：虚地址空间到物理内存的映射

线性地址 → 物理地址

分页单元（*paging unit*）把线性地址转换成物理地址。其中的一个关键任务是把所请求的访问类型与线性地址的访问权限相比较，如果这次内存访问是无效的，就产生一个缺页异常（参见第四章和第八章）。

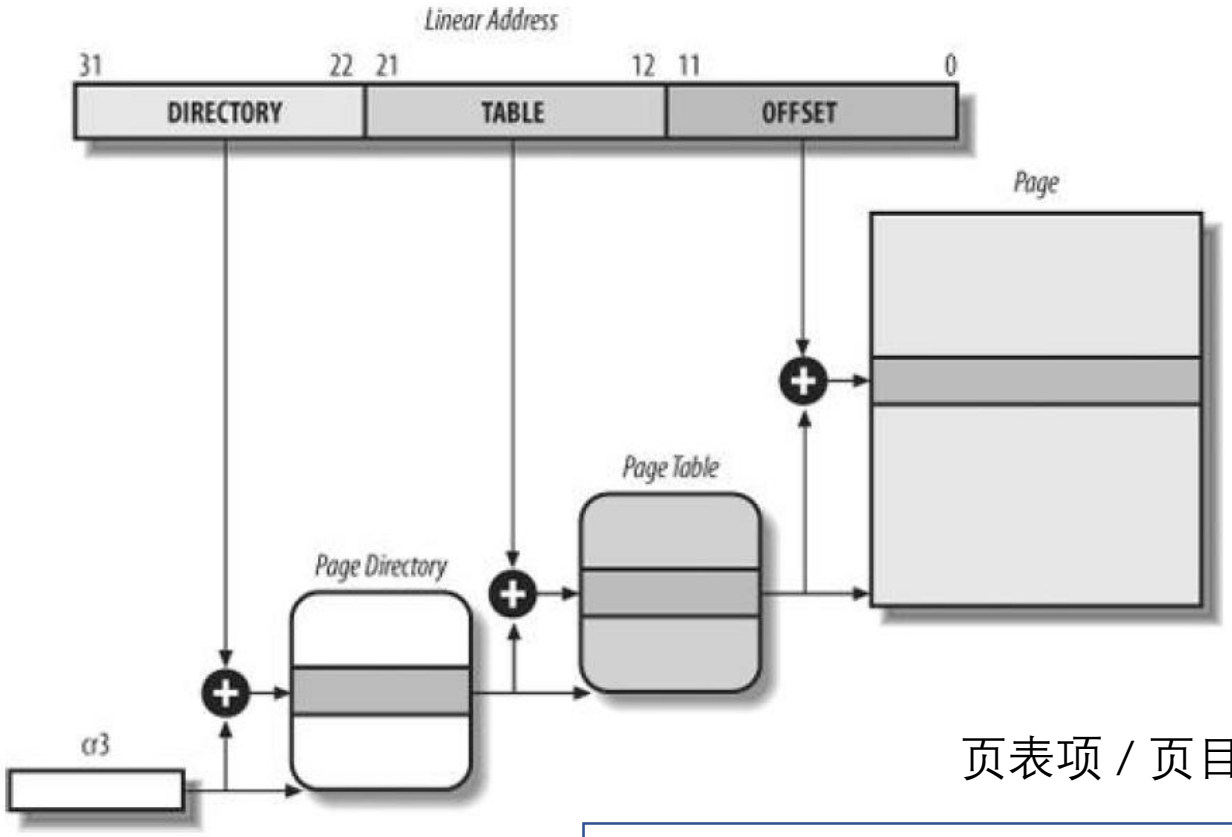
为了效率起见，线性地址被分成以固定长度为单位的组，称为页（*page*）。页内部连续的线性地址被映射到连续的物理地址中。这样，内核可以指定一个页的物理地址和其存取权限，而不用指定页所包含的全部线性地址的存取权限。我们遵循通常习惯，使用术语“页”既指一组线性地址，又指包含在这组地址中的数据。

分页单元把所有的RAM分成固定长度的页框（*page frame*）（有时叫做物理页）。每一个页框包含一个页（*page*），也就是说一个页框的长度与一个页的长度一致。页框是主存的一部分，因此也是一个存储区域。区分一页和一个页框是很重要的，前者只是一个数据块，可以存放在任何页框或磁盘中。

把线性地址映射到物理地址的数据结构称为页表（*page table*）。页表存放在主存中

线性地址（Linear Address） → 物理地址
1、常规分页　32位逻辑地址 → 32位物理地址

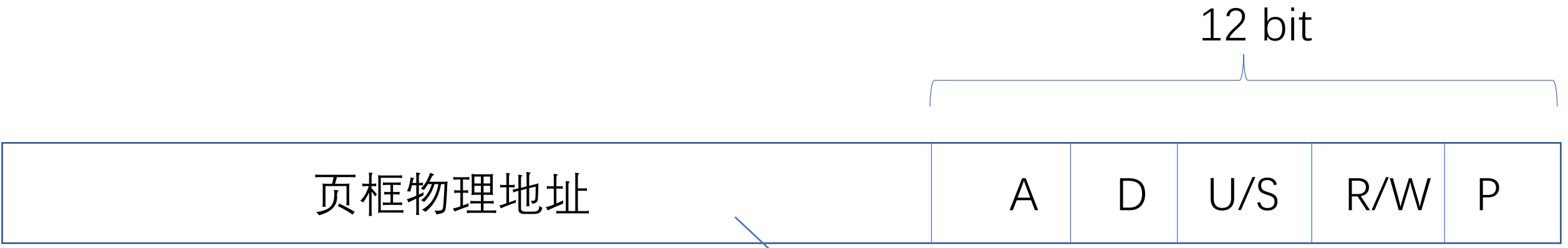
Figure 2-7. Paging by 80 x 86 processors



页表项 / 页目录项（4字节）

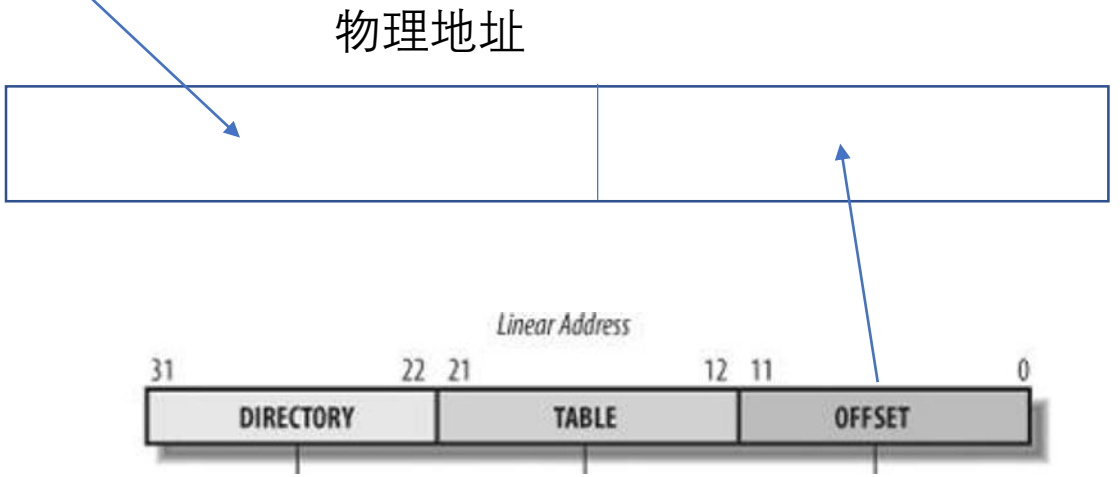
主存页框号	

PTE（Page Table Entry）：功能：　地址映射 & 内存保护



- 内存保护：地址映射前的check
  - PTE = null，或
  - CPL == 0x11 且 U/S == 0，或
  - 写操作 且 R/W == 0
  - 不通过，不可以地址映射。
  - 抛出14#缺页异常

- 地址映射

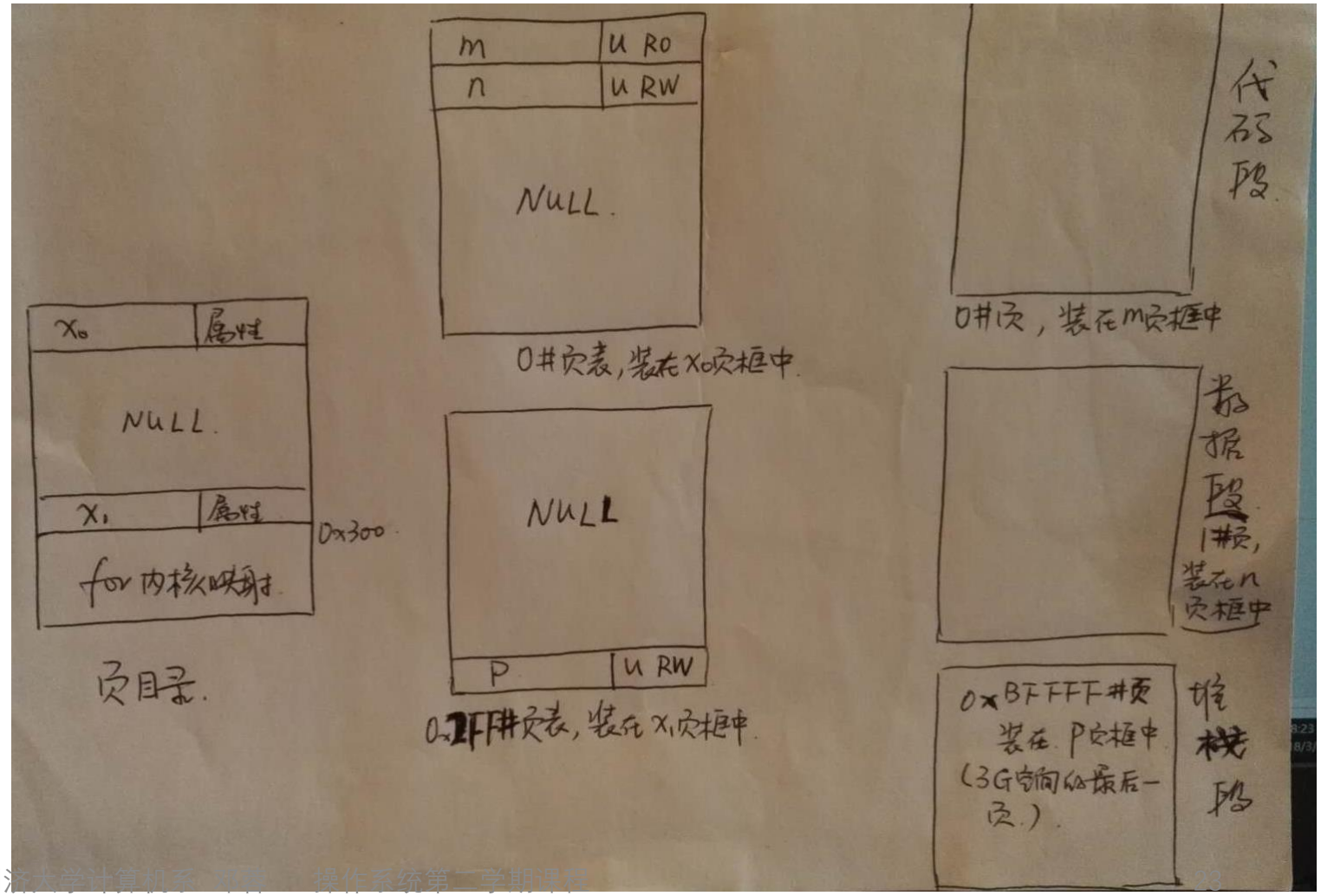


Linux的分页

- 每一个进程有它自己的页全局目录和自己的页表集。页全局目录的起始地址登记在这个进程的task_struct中。
- CPU 控制寄存器cr3的值是现运行进程页全局目录的起始地址（物理地址）
- 进程切换时，保存、恢复 cr3寄存器。

Linux的分页（常规分页）
例　一个最小的静态链接的进程的2级页表

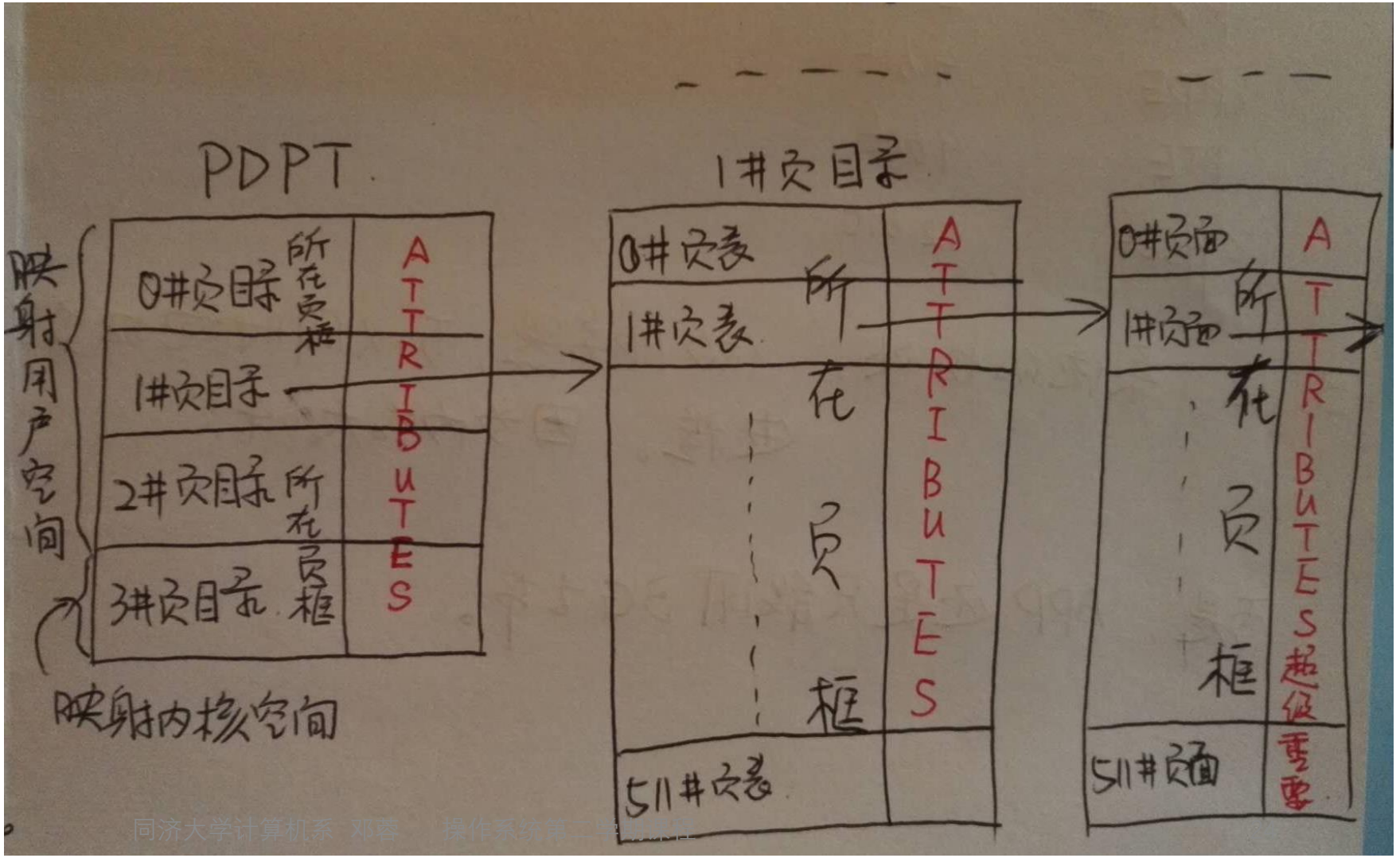
- 1页代码，1页数据，1页堆栈。
- 堆栈在3G空间的末尾。简化一下，代码起始地址是0。
- 抛出缺页异常 ~
- 地址映射　~



2、32位逻辑地址→36位物理地址
（开启PAE36机制）

逻辑地址分布
2 + 9 + 9 + 12

常规尺寸的页
4K字节



3、48位逻辑地址→36位物理地址（80X86，未开启PAE36机制）

逻辑地址分布

9 + 9 + 9 + 9 + 12

常规尺寸的页

4K字节

4级分页模型。4种页表分别称为

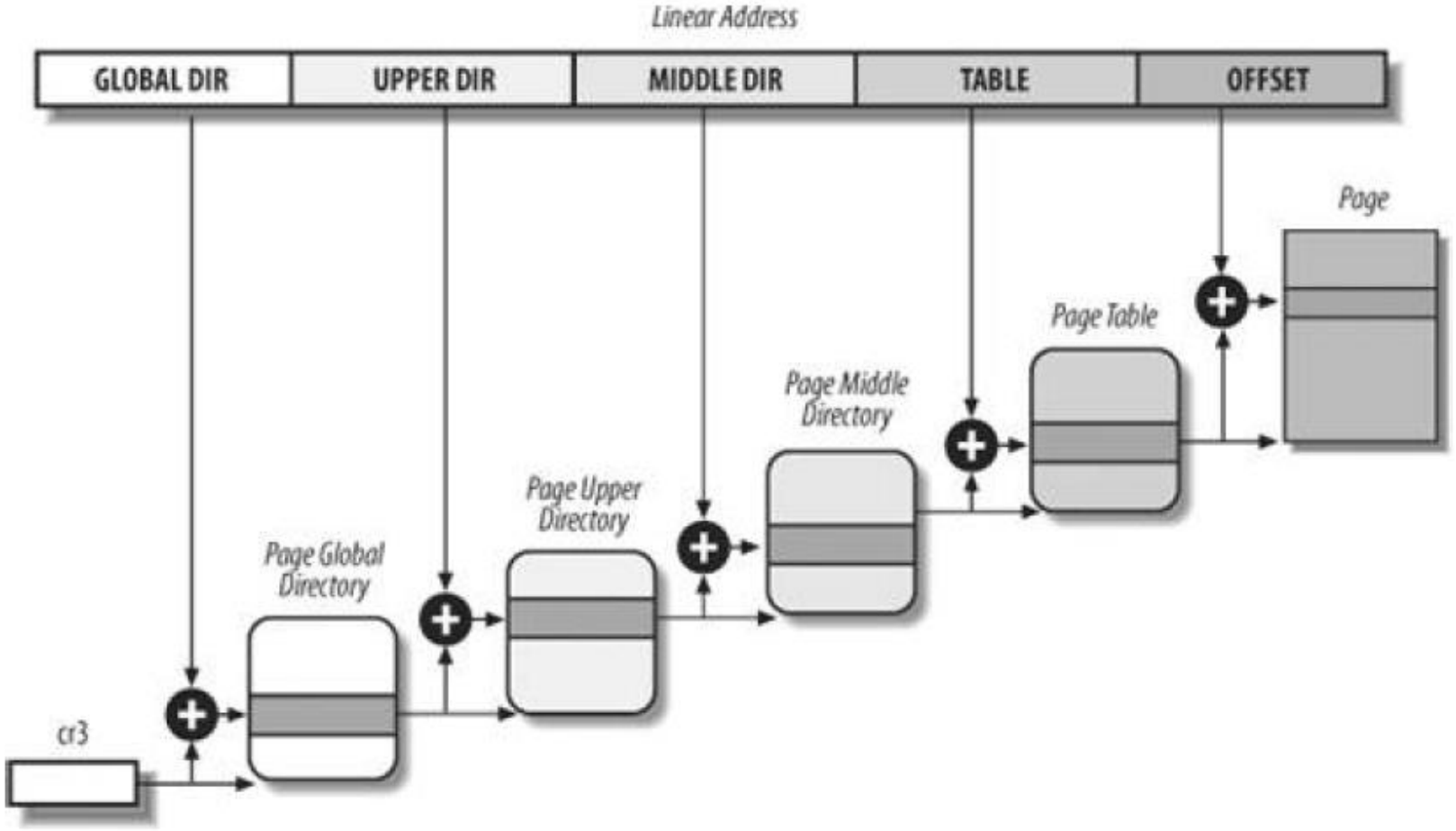
页全局目录 PGD

页上级目录 PUD

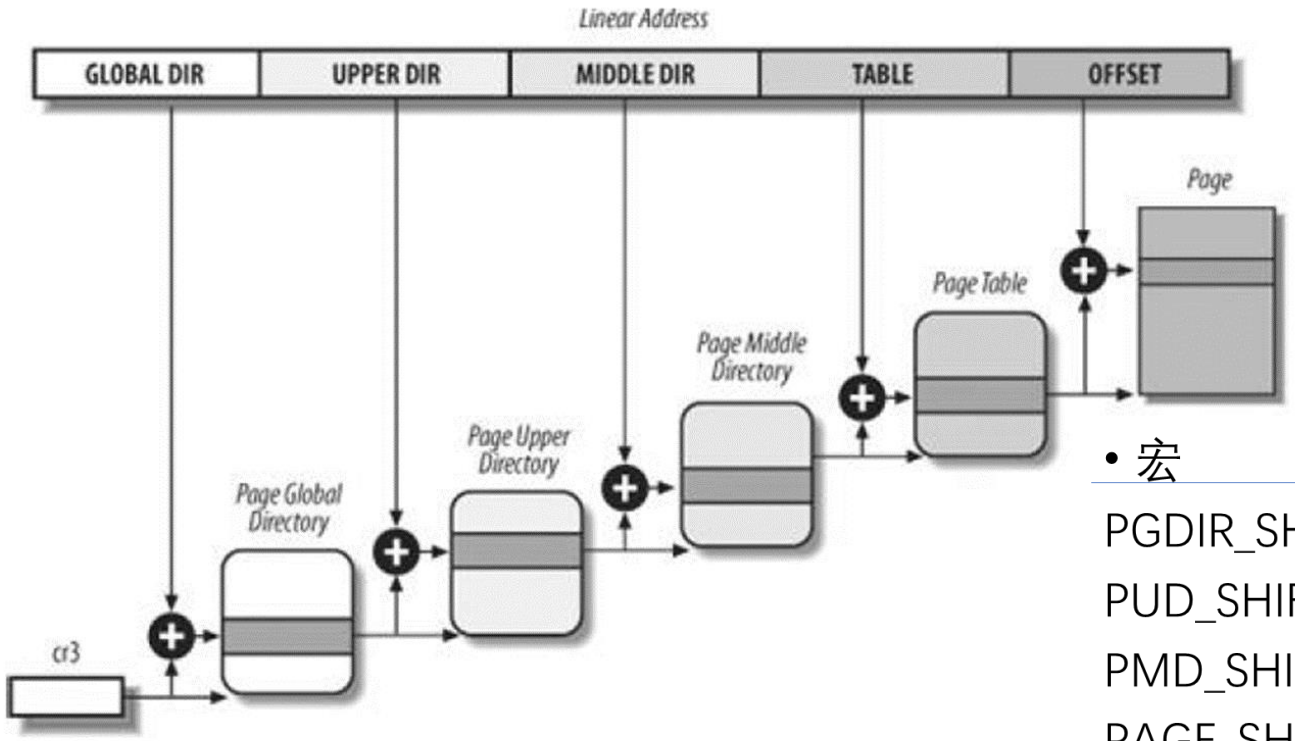
页中间目录 PMD

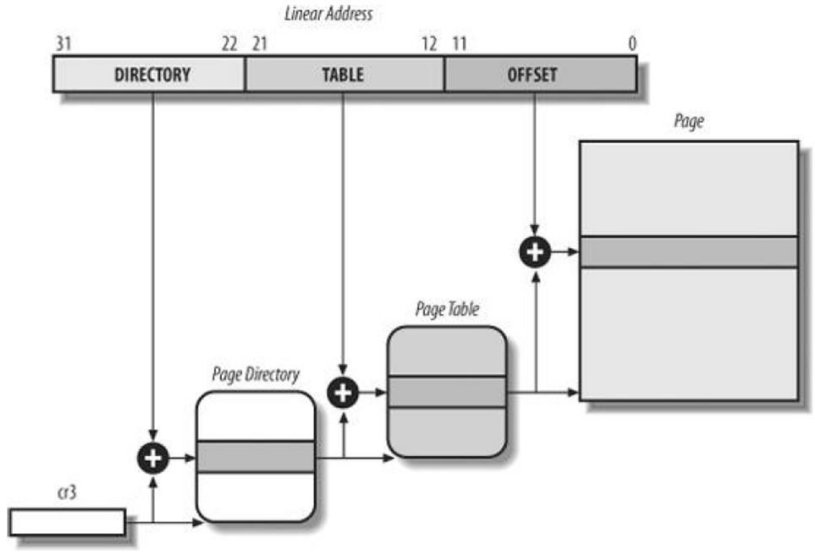
页表 PAGE

Linux 4级页表结构 for x86-64系统



4、Linux通用页表模型





• 宏	48 x86-64	32 80x86
PGDIR_SHIFT	39	22
PUD_SHIFT	30	22
PMD_SHIFT	21	22
PAGE_SHIFT	12	12

取线性地址的高位（从47bit到PGDIR_SHIFT），查页全局目录，得PGD。

比较 PGDIR_SHIFT 和 PUD_SHIFT，

- 相等，页全局目录就是继续映射需要的页上级目录。
- 不相等，PGD address指向的页框，装有继续映射需要的页上级目录。