

Linux内核体系架构

李超

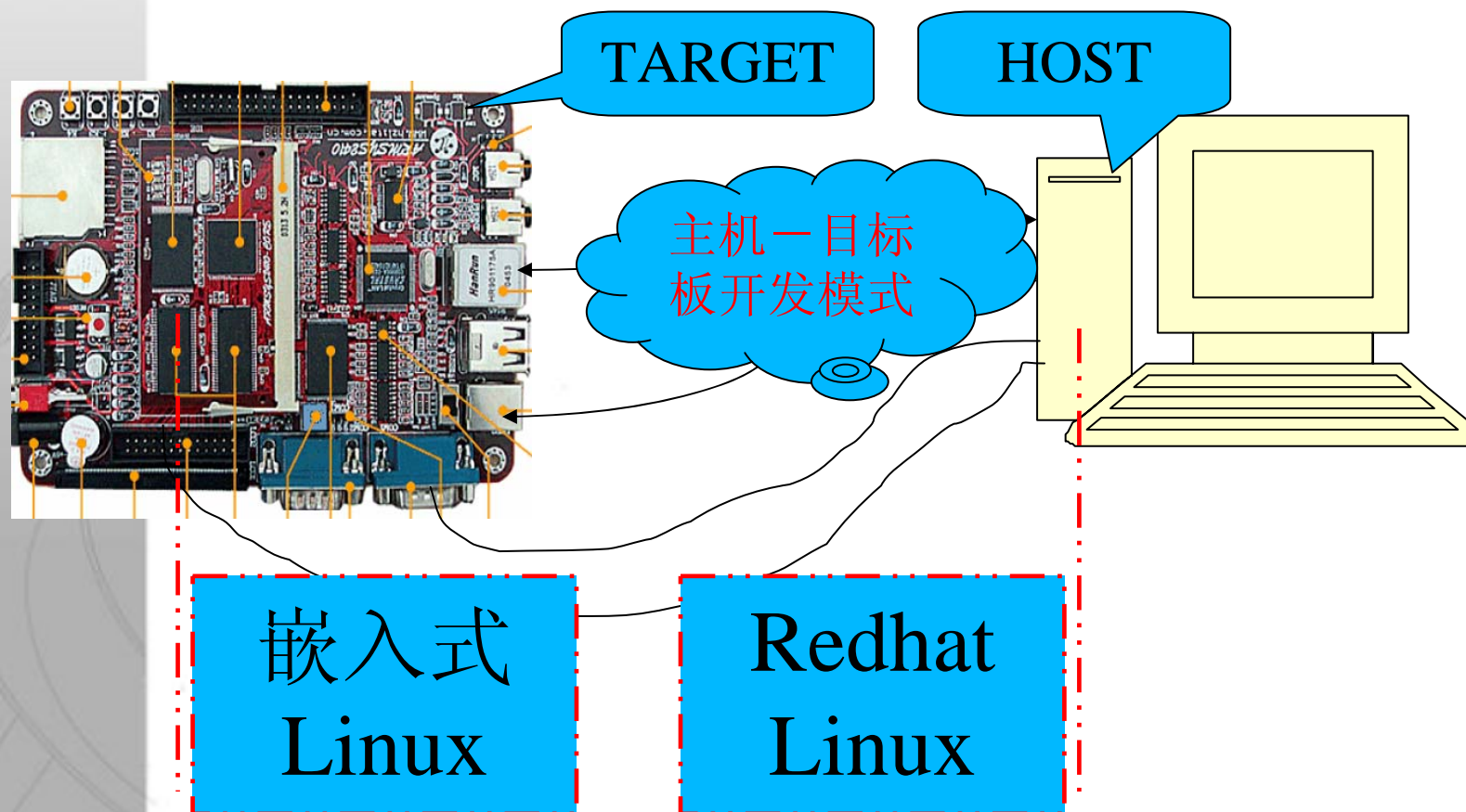
Lichao-runing@163.com


13913004799

PART ONE

构建嵌入式Linux 系统

嵌入式Linux系统开发模型



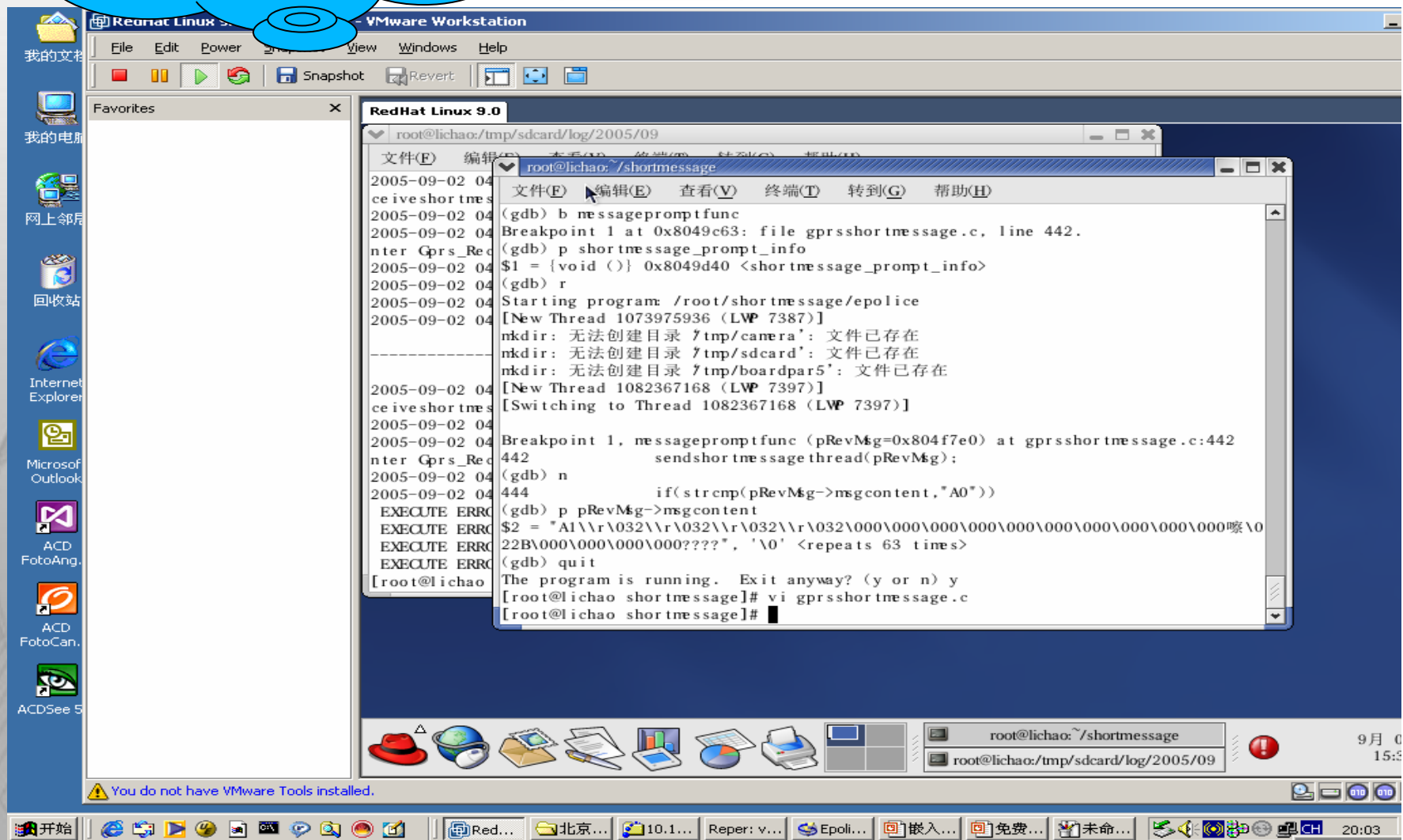


开发环境建立
(7057端)

Regnat Linux VMware Workstation

File Edit Power View Windows Help

1. 安装 Redhat Linux (推荐使用VMware)

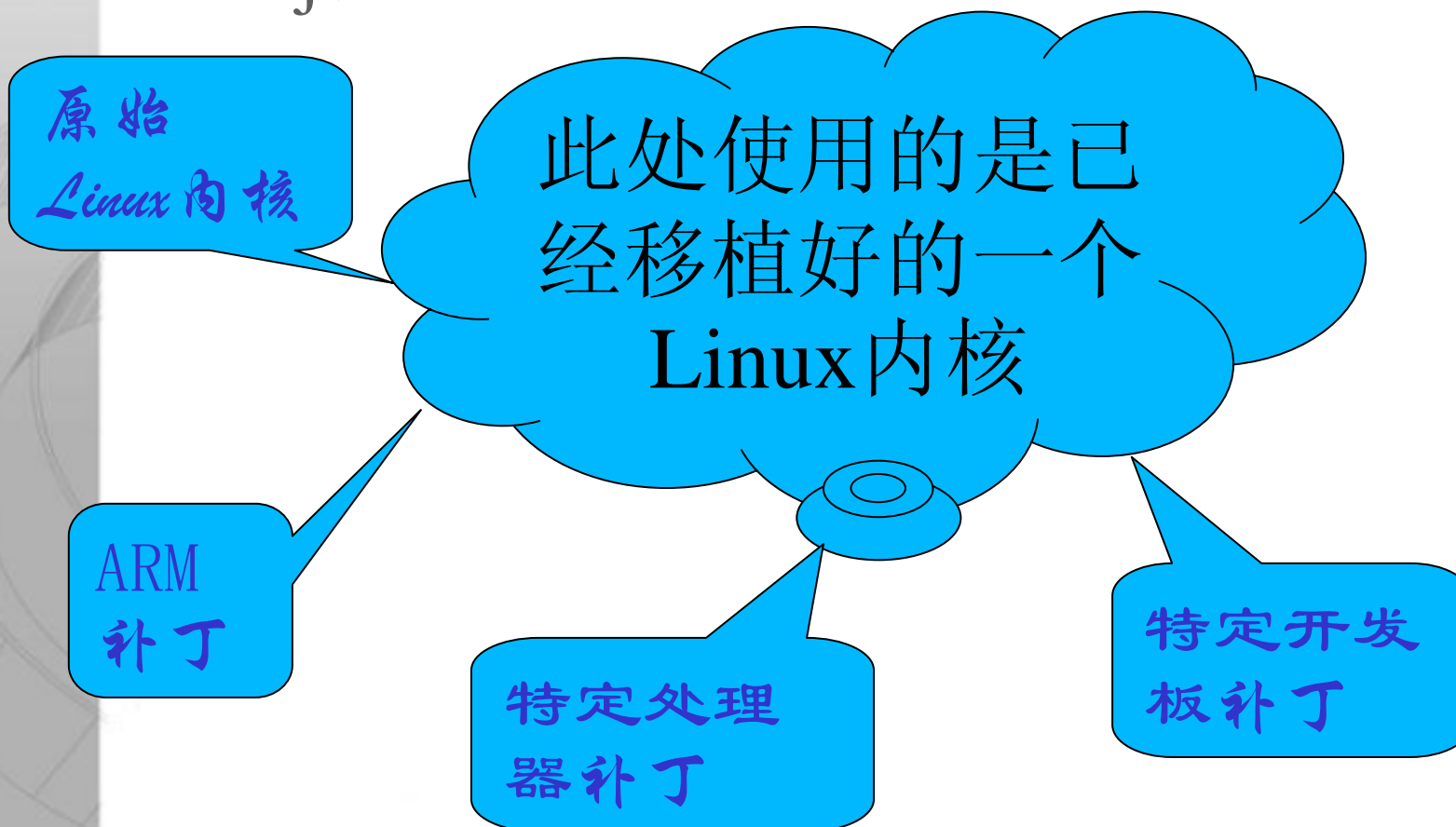


2. 安装交叉编译器

- 方法1：自己构建交叉编译器
- 方法2：使用已有的交叉编译器（针对特定处理器）
 - 下载cross-2.95.3.tar.gz
 - 建立/usr/local/arm目录
 - 在这个目录下执行命令
 - tar xzvf cross-2.95.3.tar.gz
 - 把这个目录加到PATH环境变量中

3. 安装嵌入式Linux内核

- mkdir Linux_kernel
- cd Linux_kernel
- tar xjvf linux-xx-xx.tar.bz2



4.裁減嵌入式Linux内核

➤ make menuconfig

```

Main Menu
Arrow keys navigate the menu.  <Enter> selects submenus --->.
Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
<M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help.
Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

Code maturity level options --->
Loadable module support --->
System Type --->
General setup --->
Parallel port support --->
Memory Technology Devices (MTD) --->
Plug and Play configuration --->
Block devices --->
Multi-device support (RAID and LVM) --->
Networking options --->
Network device support --->
Amateur Radio support --->
IrDA (infrared) support --->
ATA/IDE/MFM/RLL support --->
SCSI support --->
I2O device support --->
ISDN subsystem --->
Input core support --->
Character devices --->
Multimedia devices --->
File systems --->
Console drivers --->
Sound --->
Multimedia Capabilities Port drivers --->
USB support --->
Bluetooth support --->

<Select>  < Exit >  < Help >

```

5.编译嵌入式Linux内核

❖ make clean

❖ make dep

❖ make zImage

❖ 最后得到内核压缩文件zImage

6.裁减嵌入式文件系统

❖ 方案一：自己构建

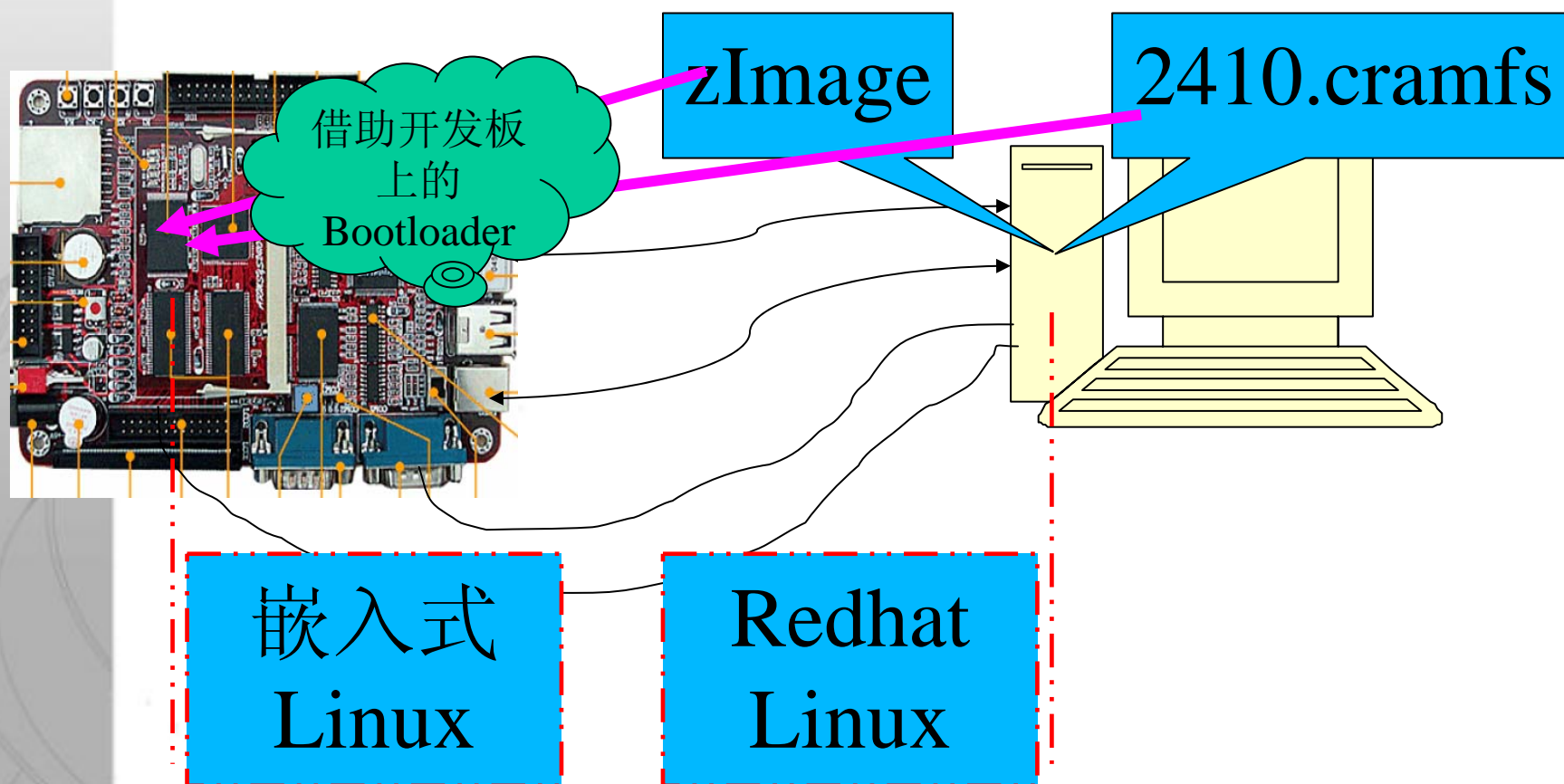
❖ 方案二：使用别人已经裁减移植好的文件系统（针对特定开发板）

❖ ***mount -o loop kk.cramfs /mnt/fs***

❖ ***cp app1.exe /mnt/fs***

❖ ***mkcramfs /mnt/fs 2410.cramfs***

7. 更新目标板上的系统



嵌入式Linux内核 总论

典型Linux系统构成图

指那些向用户提供的服务被看作是操作系统部分功能的程序

是指那些字处理程序、Internet 浏览器程序或用户自行编制的各种应用程序

由CPU、内存、I/O、硬盘等底层硬件设备构成

用户应用程序

操作系统服务

操作系统内核

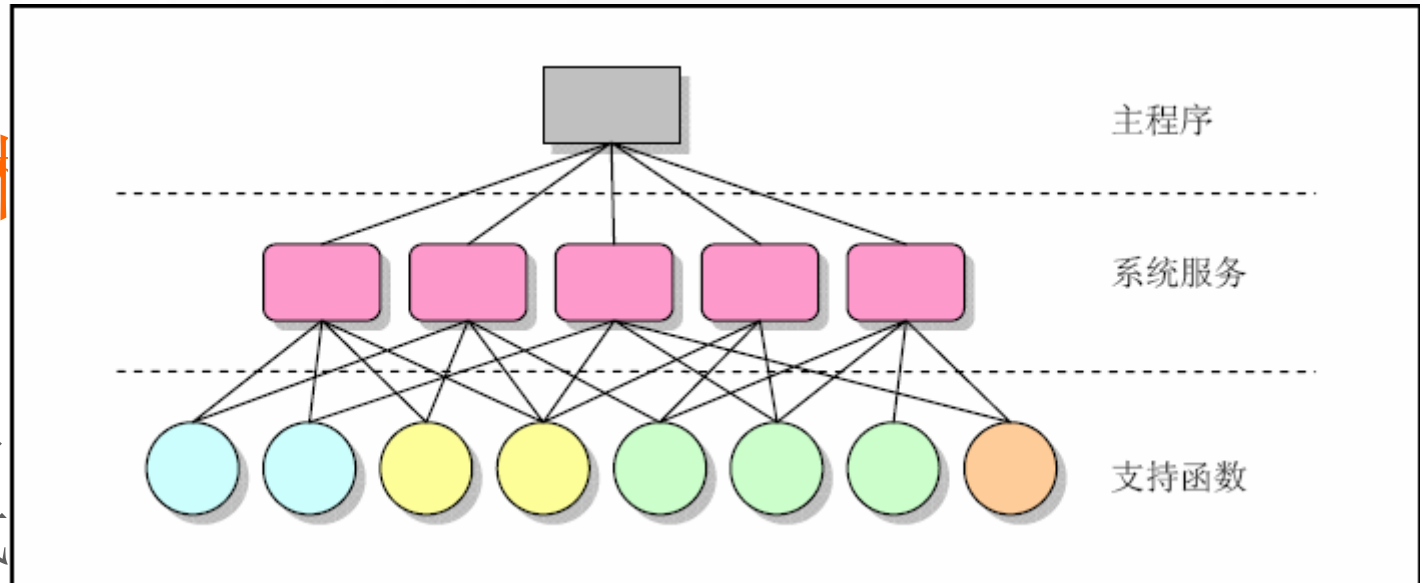
硬件系统

与计算机硬件进行交互，实现对硬件部件的编程控制和接口操作，调度对硬件资源的访问，并为计算机上的用户程序提供一个高级的执行环境和对硬件的虚拟接口。

1.Linux内核

❖ 操作系统内 式和层次式

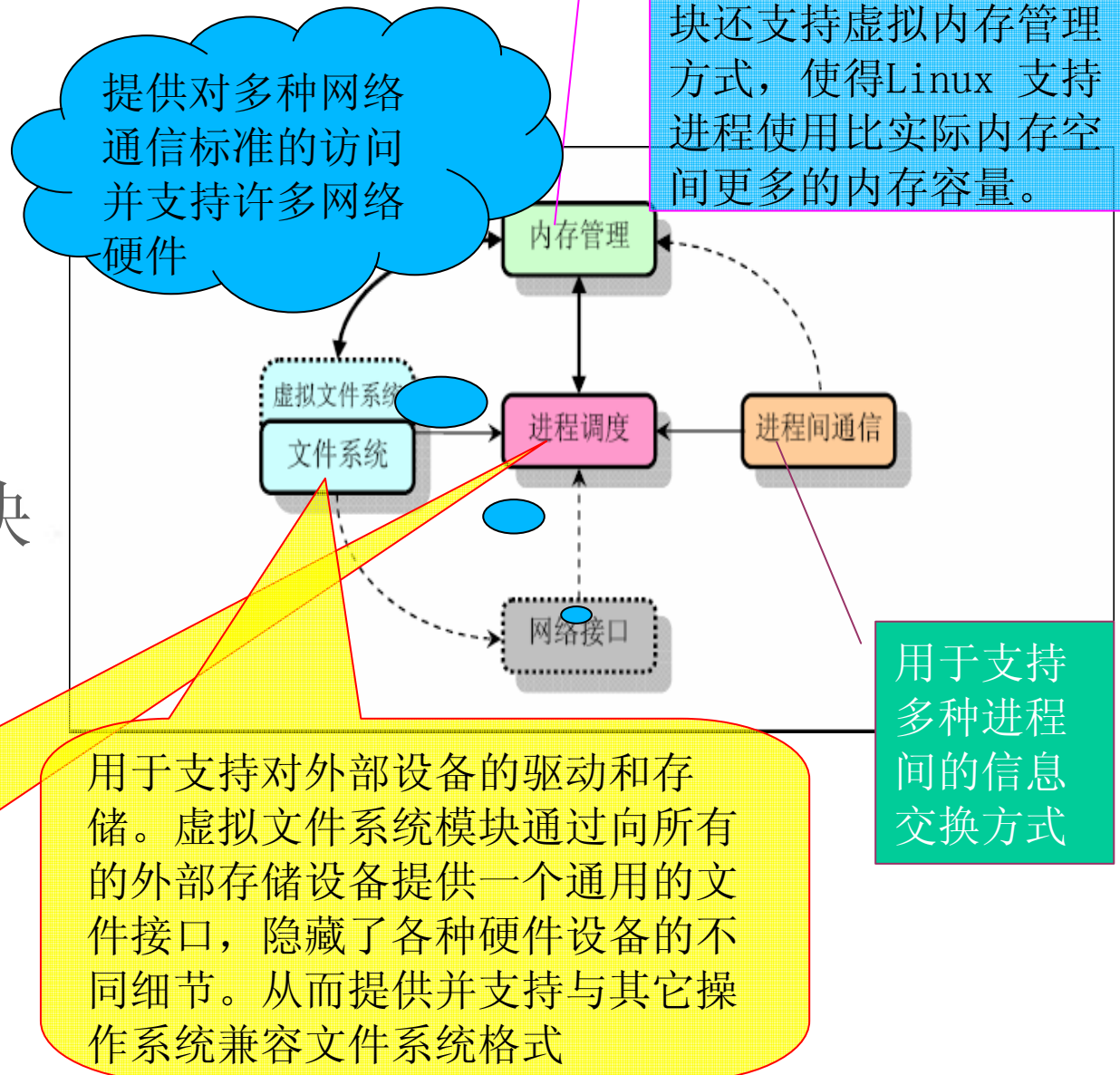
- ❖ 在单内核模式的系统中，操作系统所提供服务的流程为：应用主程序使用指定的参数值执行系统调用指令 (int x80)，使CPU从用户态（User Mode）切换到核心态（Kernel Model），然后操作系统根据具体的参数值调用特定的系统调用服务程序，而这些服务程序则根据需要再底层的一些支持函数以完成特定的功能。在完成了应用程序所要求的服务后，操作系统又从核心态切换回用户态，返回到应用程序中继续执行后面的指令

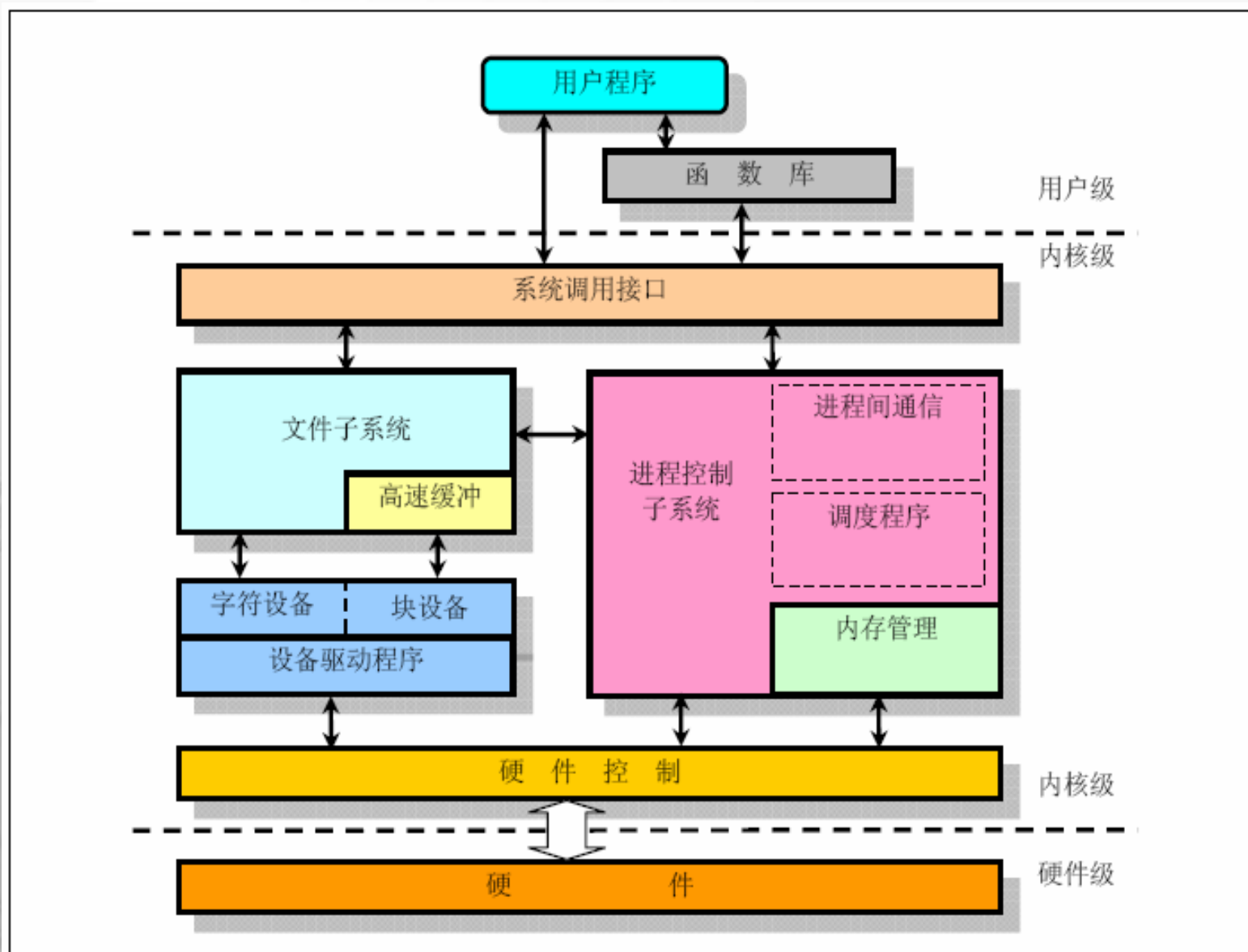


单内核模式的简单结构模型

2.Linux 内核系统模块体系结构

- ❖ 进程调度模块
- ❖ 内存管理模块
- ❖ 文件系统模块
- ❖ 进程间通信模块
- ❖ 网络接口模块





内核结构框图

嵌入式内核知识点讲解大纲

- ❖ 内存管理
- ❖ 进程
- ❖ 中断、异常和系统调用
- ❖ 文件系统
- ❖ 进程间通信
- ❖ 设备驱动
- ❖ 信号

PART TWO

Linux 内存管理

提纲

- ❖ ARM-LINUX内存管理
- ❖ X86下LINUX内存管理

ARM-LINUX内存管理

ARM内存管理

- 虚拟内存管理
- 物理内存管理
- 虚拟地址到物理地址的映射

内存管理的本质特征

- 在ARM-Linux系统中，一个可执行程序被加载到内存后就成为一个进程。内存是每一个进程的“生活场所”和“表演舞台”。在嵌入式开发系统中，由于受到成本、体积和功耗等因素的制约，内存都非常的小，一般都为64MB、32MB，甚至更小。这就可能导致下面几个场景发生：
 - 其一、运行某个进程时，机器内存容纳不下该进程所有的代码、数据和堆栈，而只能容纳其一部分。
 - 其二、对于运行多任务系统来说，如果两个进程所运行的空间重叠，这两个进程将无法同时运行，否则一个进程可能破坏另一个进程的“表演舞台”。

解决问题思路 1

- 它大实及使几，用它中内的想占内存间的一个栈时据的一时堆段数换：段和一或置买一据同（面事意数在码页的任、且代中本在码而间理基，代，空管个程用分内存一进便部围。内样的会一范高。了这存只的一最入于内上小某也引基量际其用率
- 操作部的内除计裁然中入清设加仍存调分样只则内它部这内存分在把那可间它到系统的我们时其用存透明，我一而要作存，某，需操内，事实在中当，由已经操作，的使内存上时把已操作，面，到盘分同。这些操作基作分存某存除

解决问题思路 2

- 为了解决多个进程运行空间重叠的矛盾，在ARM-Linux系统中，引入了虚拟内存这个概念。
- 虚拟内存概念是相对于物理内存来说地，它并不存在，而且非常巨大，通常为4GB(要根据地址线来决定)大小。
- 每一进程都有一套独立于其它进程的虚拟地址空间，当hello1和hello2可执行程序被加载到内存中时，将会被MMU部件映射到不同的物理地址空间中去，这样它们即使同时运行，也就不会发生冲突了。

进程与虚拟内存管理

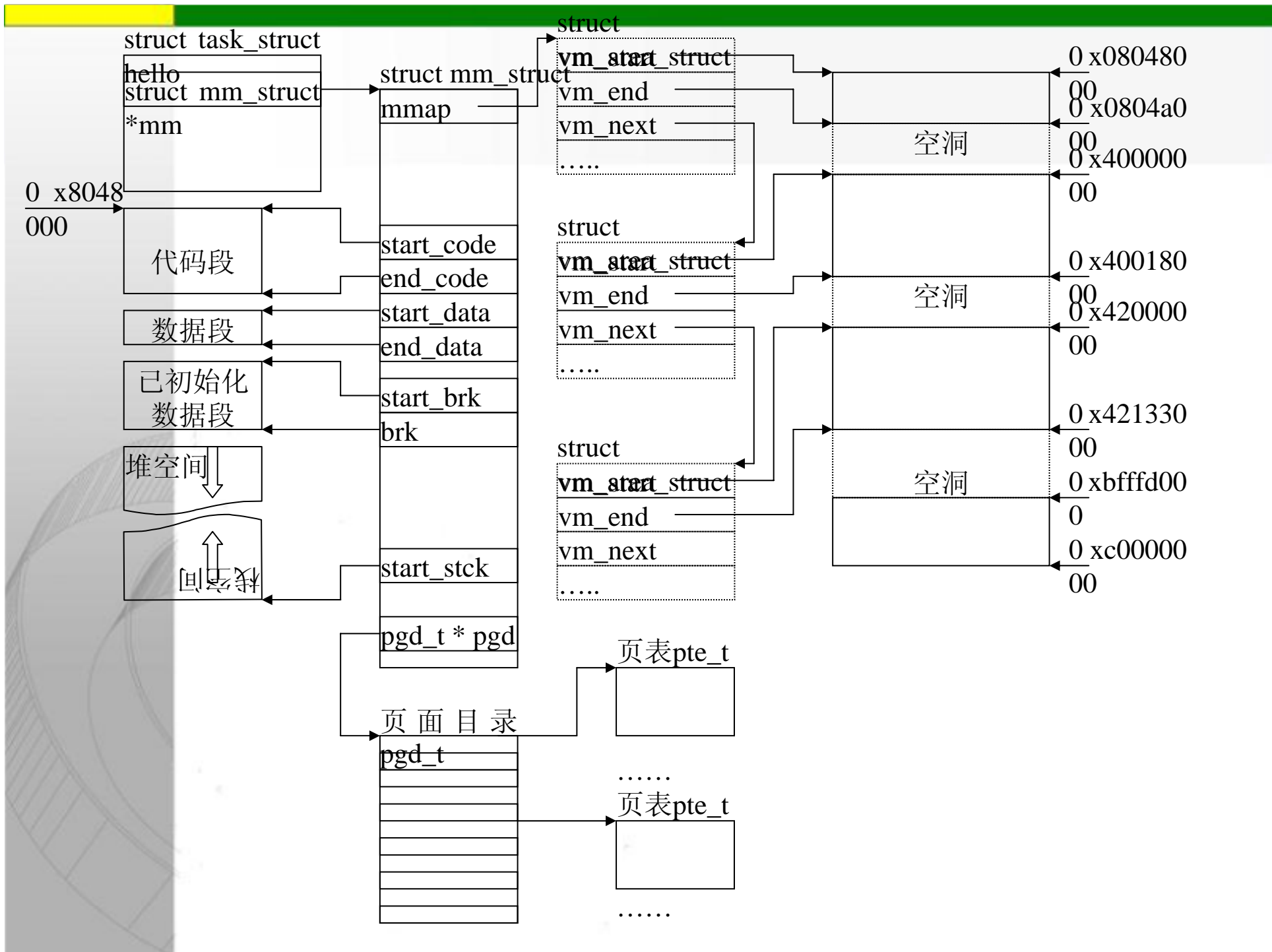
- `#include <stdio.h>`
- `void main()`
- `{`
- `while(1)`
- `printf("hello world\n");`
- `}`

cat命令察看该进程所使用到的虚拟内存空间

- `#cat /proc/<pid>/maps`
- `[root@localhost 2772]# cat maps`
- `08048000-08049000 r-xp 00000000 08:02 755919 /example/linux-kernel/chapter4/hello`
- `08049000-0804a000 rw-p 00000000 08:02 755919 /example/linux-kernel/chapter4/hello`
- `40000000-40015000 r-xp 00000000 08:02 97988 /lib/ld-2.3.2.so`
- `40015000-40016000 rw-p 00014000 08:02 97988 /lib/ld-2.3.2.so`
- `40016000-40018000 rw-p 00000000 00:00 0`
- `42000000-4212e000 r-xp 00000000 08:02 816069 /lib/tls/libc-2.3.2.so`
- `4212e000-42131000 rw-p 0012e000 08:02 816069 /lib/tls/libc-2.3.2.so`
- `42131000-42133000 rw-p 00000000 00:00 0`
- `bfffd000-c0000000 rwxp ffffe000 00:00 0`
- 格式：
- 虚拟内存开始地址—结束地址 访问权限 偏移 主设备号：次设备号 i节点 文件

hel10进程占用的虚拟内存空间图





虚拟内存管理

- 用户空间和内核空间
- 在ARM-Linux系统中，每个进程都有一套独立与其它进程的虚拟地址空间，这个地址空间有4GB大小，其中0~3GB可以被其任意使用，我们称之为用户空间，而3GB~4GB为内核虚拟空间，所有的进程的虚拟空间中，3GB~4GB部分都是相同的，这部分空间是普通进程不能随意使用的。

虚拟内存段数据结构描述

- 一个进程在运行时并不可能使用到如此大的虚拟地址空间，它们仅会使用其中的部分空间，而且并不一定连成一片，可能会被分割成几块，每一块连续的空间被称为虚拟内存段，在ARM-Linux系统中，虚拟内存段用struct vm_area_struct结构体来描述，所有的虚拟内存段通过vm_area_struct结构体中指针成员构成链表

红黑树

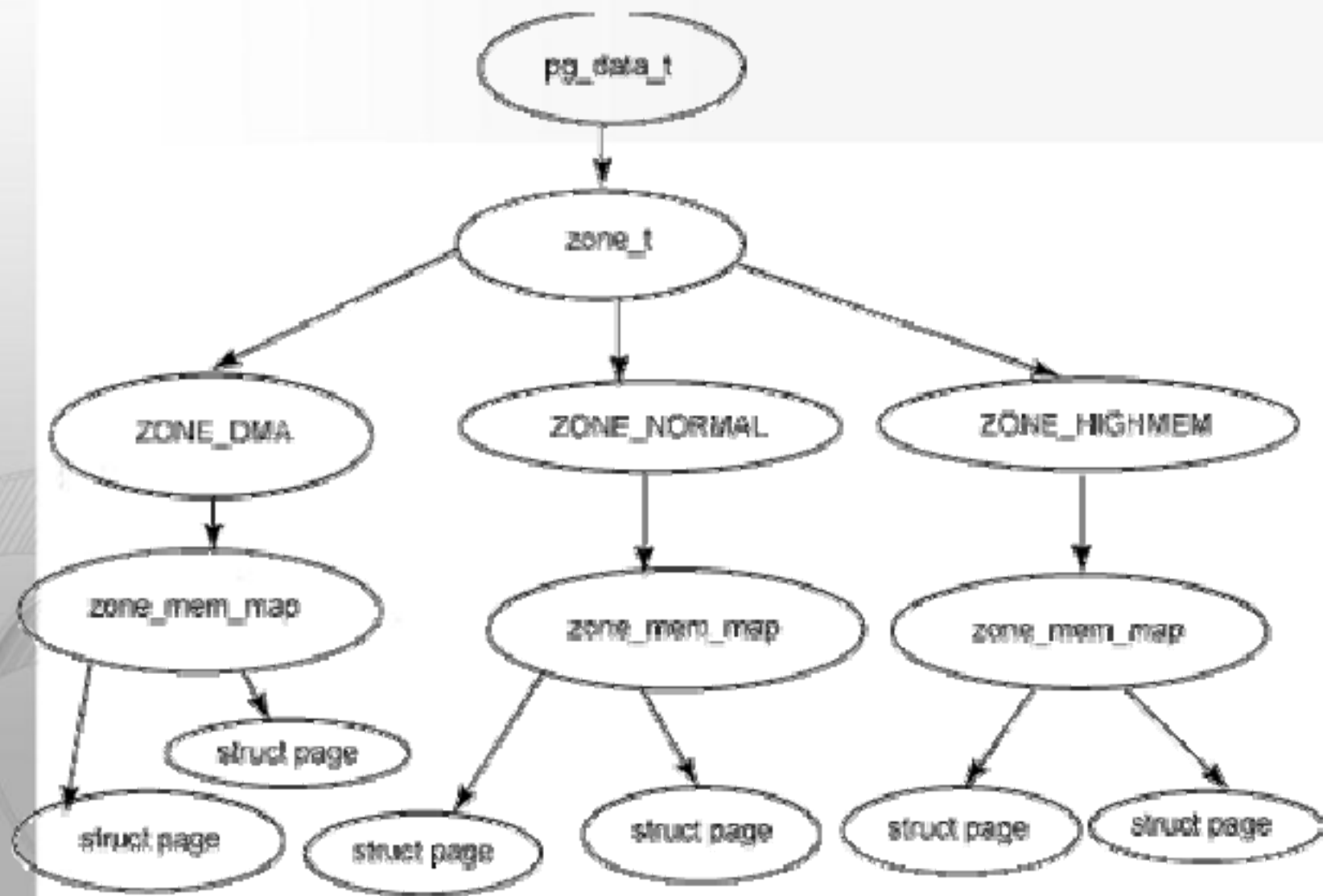
- 为了更好的操作vma段，在ARM-Linux系统中将这些虚存段组织成一棵红黑树。红黑树中有三种类型的节点：
- 根节点：数据结构类型为`rb_root_t`，指向红黑树的根
- 普通节点：数据结构类型为`rb_node_t`
- 叶子节点：数据结构类型为`struct vm_area_struct`，代表一个具体的虚存段

虚拟内存分页机制

- 在Linux系统中虚拟内存非常大，有4GB的空间。上一节中描述了将连续在一起的虚拟内存组成虚拟段，用红黑树来进行管理，但是虚存段仍然很大，使用起来及其不方便。为了更好的使用虚拟内存，在ARM-Linux系统中采用分页技术。所谓分页技术就是将虚拟地址空间划分成页（page），页包含了一段连续的虚拟地址空间，页的大小一般为4KB，但也可以是1KB、64KB等大小
- 虚存中的页和物理内存中的页帧相对应，两者必须相同，而在Linux

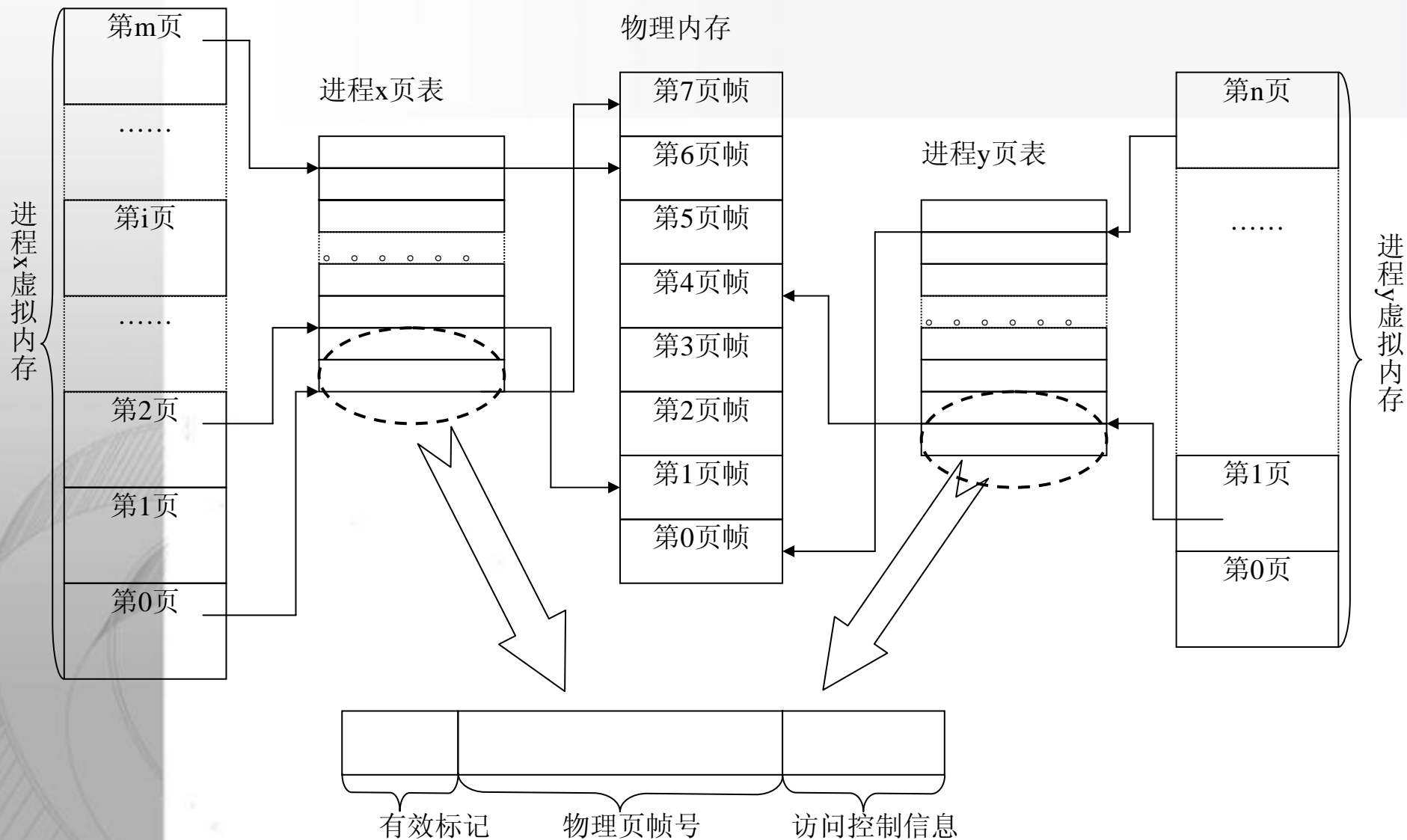
物理内存管理

- 节点的概念在内核中是使用 `struct pglist_data` 结构来实现的，它表示在ARM-Linux系统中一段连续的、均匀的以及访问速度相同的物理内存空间被。
- 一个节点的内存段按照用途可以被分成很多个区域，通常为`ZONE_DMA`、`ZONE_NORMAL`和`ZONE_HIGHMEM`区域，区域是使用 `struct zone_struct` 结构来描述的。
- 在ARM-Linux系统中，物理内存被分成一个个大小为4KB(也可能是其它值)的物理页帧，所以每个区域就是有不同数目的物理页帧构成，物理页帧使用 `struct page` 结构来表示，所有这些 `struct` 都保存在全局结构数组 `struct mem_map`



虚存到物理内存的映射

- 进程运行的空间为虚拟地址空间，但是它的“身体”却存放在物理地址空间内，因此必须在虚拟空间和物理地址空间架起一座沟通的桥梁，这座桥梁就是地址映射。
- 虚拟地址空间简称虚存，它采用页面来组织，也就是说，页面是虚存操作的最小单位，通常情况下页面可以是1KB、4KB、64KB甚至更大，此处以4KB大小为例。
- 虚地址由两个部分构成：页号和页内偏移量。如果页的大小设置为4KB，则虚拟地址的低12位表示页内偏移量，高20位表示页号。
- 同样物理地址空间也采用页面来组织，不过称为页帧（page frame），大小同虚拟空间的页



例：

```
//aa.c
#include <stdio.h>
void printhello()
{
    printf("hello,world!\n");
}
void main()
{
    printhello();
}
```

编译：gcc a.c -o aa

- objdump -d aa

08048328 <printhello>:

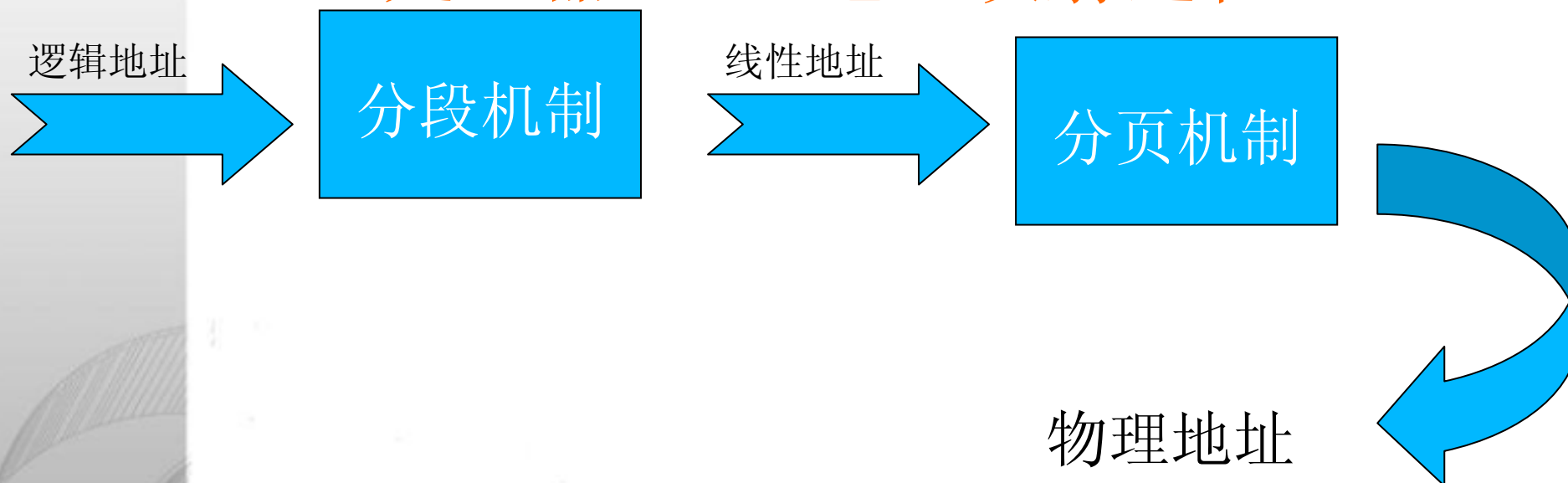
| | | | |
|----------|----------------|-------|----------------------|
| 8048328: | 55 | push | %ebp |
| 8048329: | 89 e5 | mov | %esp,%ebp |
| 804832b: | 83 ec 08 | sub | \$0x8,%esp |
| 804832e: | 83 ec 0c | sub | \$0xc,%esp |
| 8048331: | 68 04 84 04 08 | push | \$0x8048404 |
| 8048336: | e8 2d ff ff ff | call | 8048268 <_init+0x38> |
| 804833b: | 83 c4 10 | add | \$0x10,%esp |
| 804833e: | c9 | leave | |
| 804833f: | c3 | ret | |

08048340 <main>:

| | | | |
|----------|----------------|-------|----------------------|
| 8048340: | 55 | push | %ebp |
| 8048341: | 89 e5 | mov | %esp,%ebp |
| 8048343: | 83 ec 08 | sub | \$0x8,%esp |
| 8048346: | 83 e4 f0 | and | \$0xfffffffff0,%esp |
| 8048349: | b8 00 00 00 00 | mov | \$0x0,%eax |
| 804834e: | 29 c4 | sub | %eax,%esp |
| 8048350: | e8 d3 ff ff ff | call | 8048328 <printhello> |
| 8048355: | c9 | leave | |
| 8048356: | c3 | ret | |
| 8048357: | 90 | nop | |

I

ARM处理器MMU地址映射过程



- ❖ MMU 把CPU 产生的虚拟地址转换成物理地址去访问外部存储器，同时继承并检查访问权限。地址转换有四条路径。路径的选取由这个地址是被标记成段
- ❖ 映射访问还是页映射访问确定。页映射访问可以是、大、小和微页的访问。

(续)

❖ 存在主存储器内的转换表有两个级别:

❖ 第一级表 存储段转换表和指向第二级表的指针;

❖ 第二级表 存储大页和小页的转换表, 一种类型的第二级表存储微页转换表。

❖ MMU 支持基于段或页的存储器访问:

❖ 段 (Section) 构成1MB 的存储器块

默认采用此种方式

❖ 支持3 中不同的页尺寸:

➢ 微页 (Tiny page) 构成1KB 的存储器块

➢ 小页 (Small page) 构成4KB 的存储器块

➢ 大页 (Large page) 构成64KB 的存储器块

虚拟地址



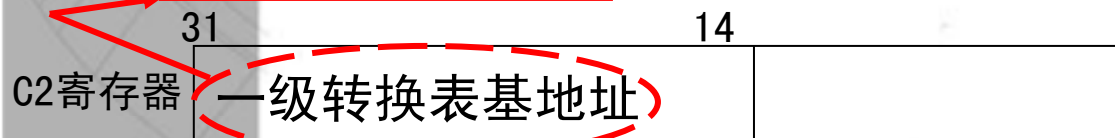
第一级描述符表



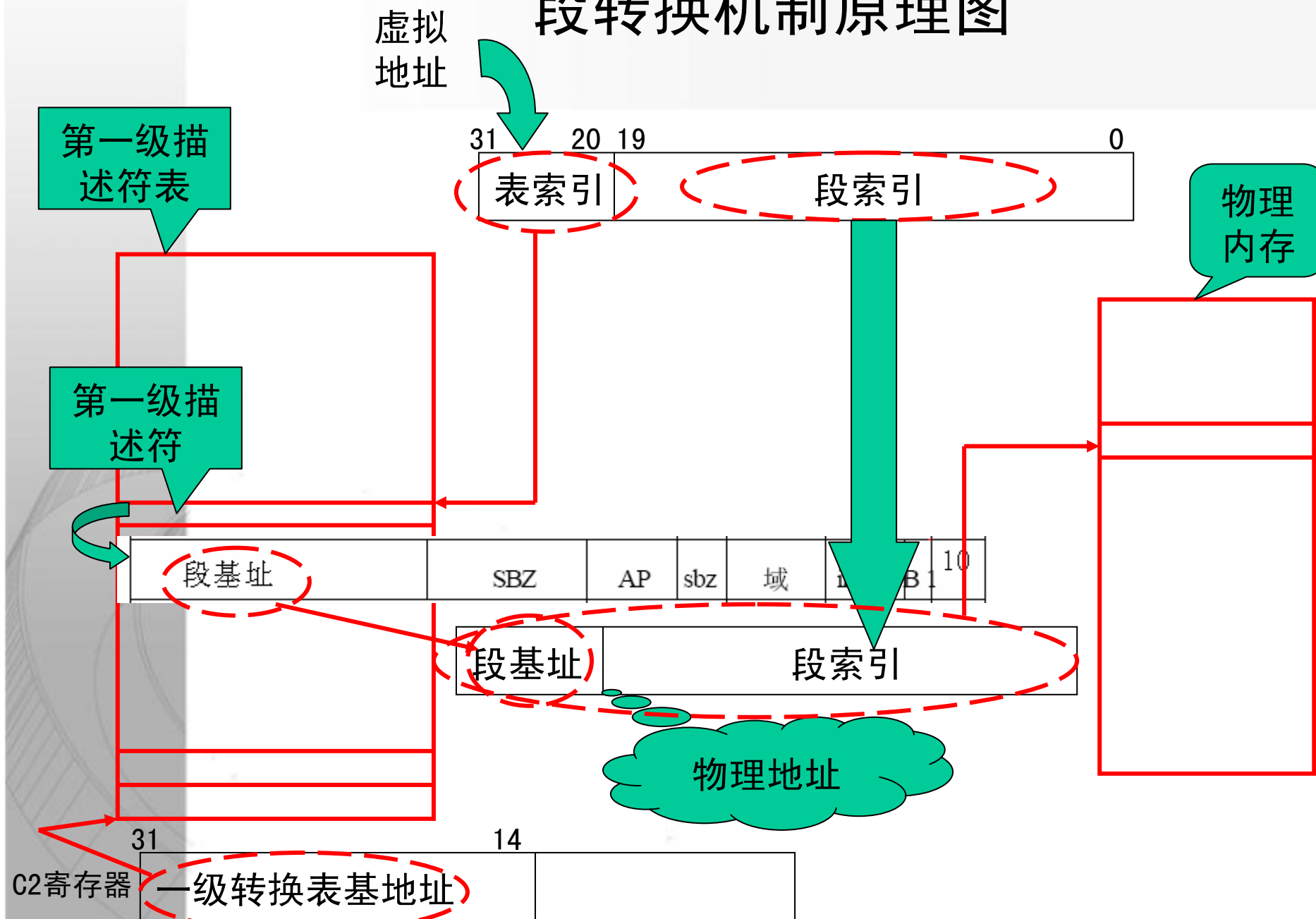
第一级描述符

第一级描述符格式

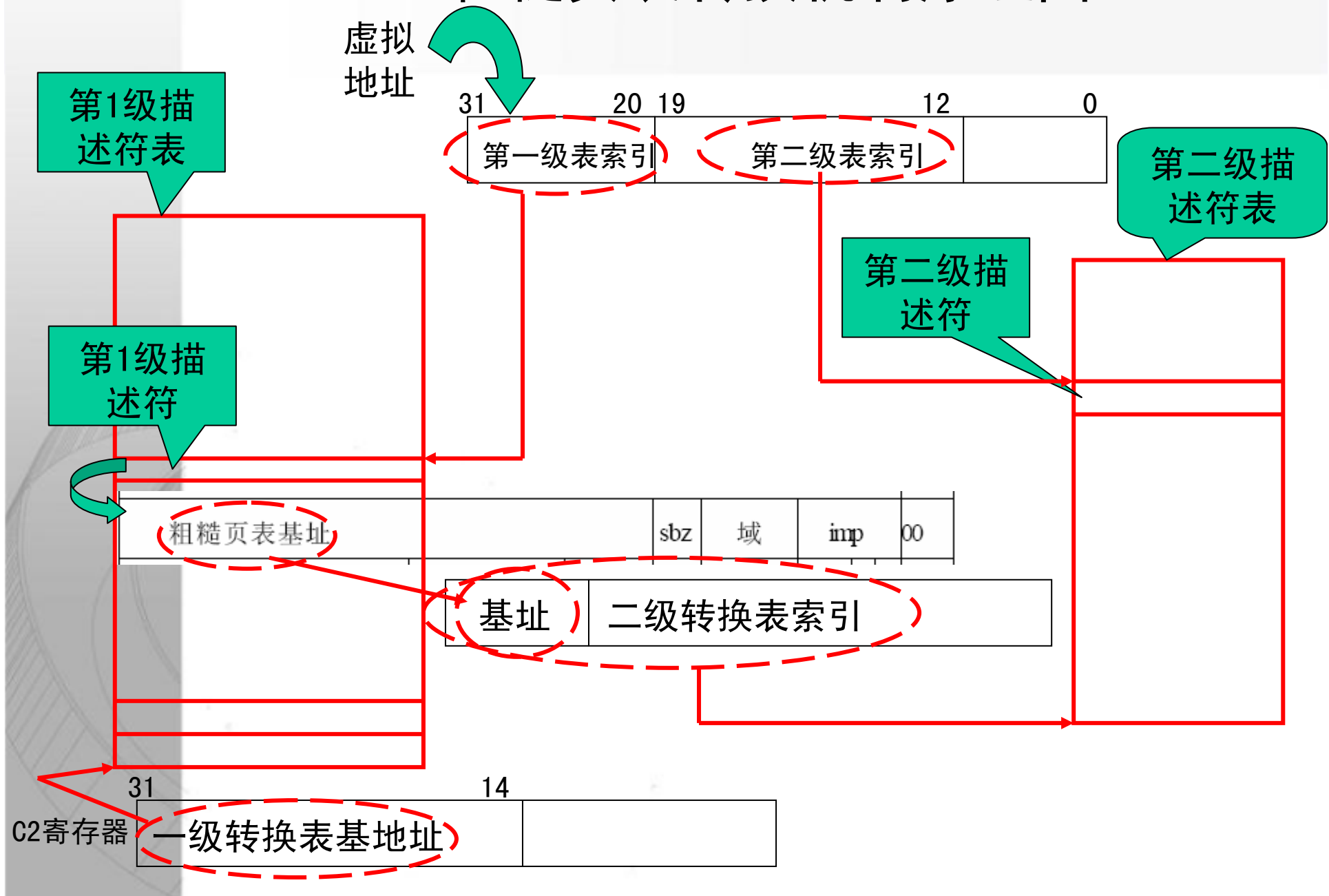
| | 31 | 20 | 19 | 12 | 11 | 10 | 9 | 8 | 5 | 4 | 3 | 2 | 10 | |
|------|--------|----|----|----|-----|----|-----|-----|---|---|-----|----|----|----|
| 错 | 忽略 | | | | | | | | | | | | | 00 |
| 粗糙页表 | 粗糙页表基址 | | | | | | | sbz | 域 | | inp | | 01 | |
| 节 | 段基址 | | | | SBZ | | AP | sbz | 域 | | inp | CB | 10 | |
| 精细页表 | 精细页表基址 | | | | | | SBZ | | 域 | | inp | | 11 | |



段转换机制原理图



粗糙页表转换机制原理图



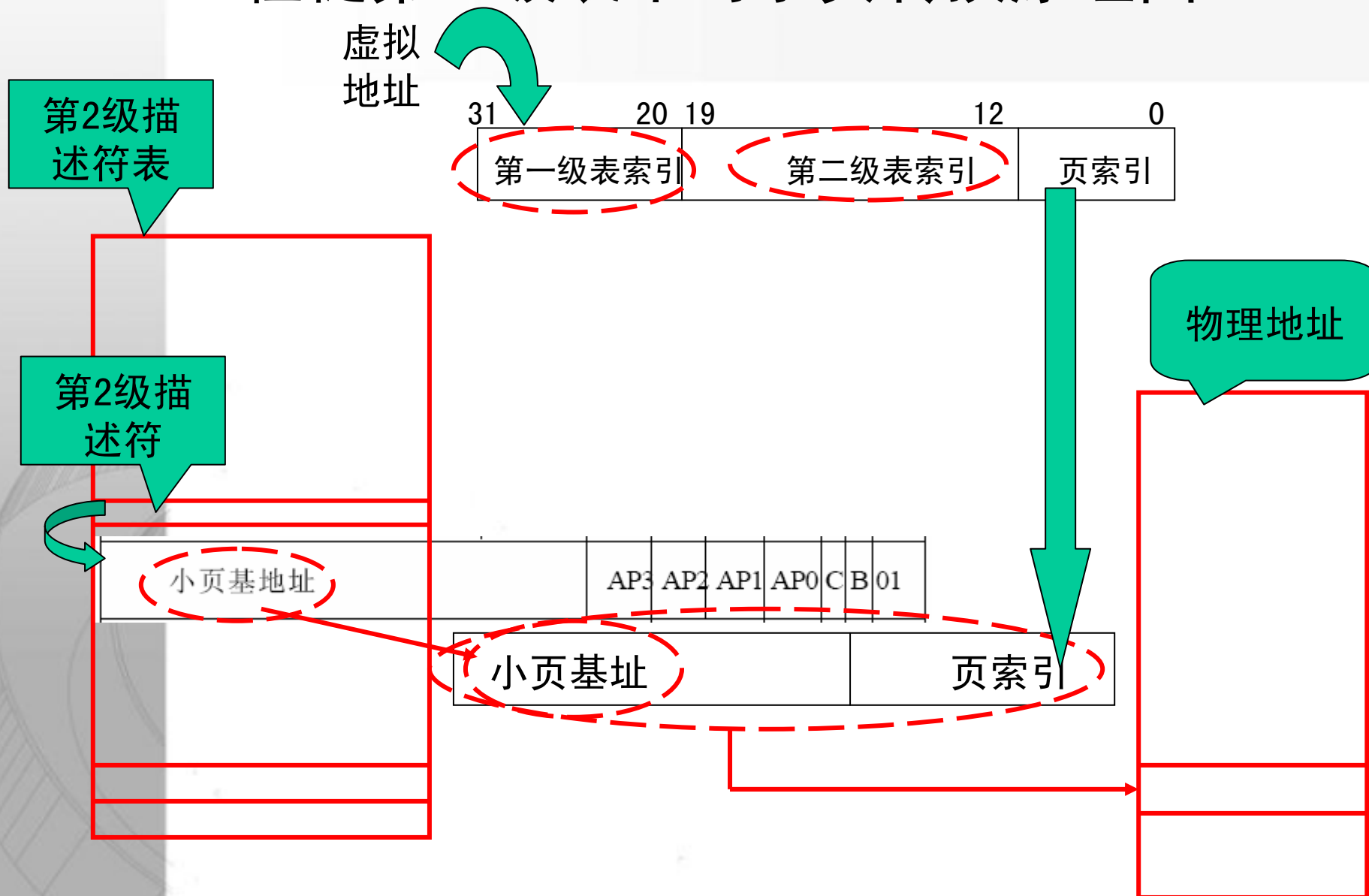
第二级描述符的格式

| | 31 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|-------|----|----|----|----|-----|-----|-----|-----|-----|-----|----|----|----|----|----|
| 错 | 忽略 | | | | | | | | | | | | | | | 00 |
| 大页 | 大页基地址 | | | | | SBZ | | AP3 | AP2 | AP1 | AP0 | C | B | 01 | | |
| 小页 | 小页基地址 | | | | | | AP3 | AP2 | AP1 | AP0 | C | B | 01 | | | |
| 微页 | 微页基地址 | | | | | | | | SBZ | | | AP | C | B | 11 | |

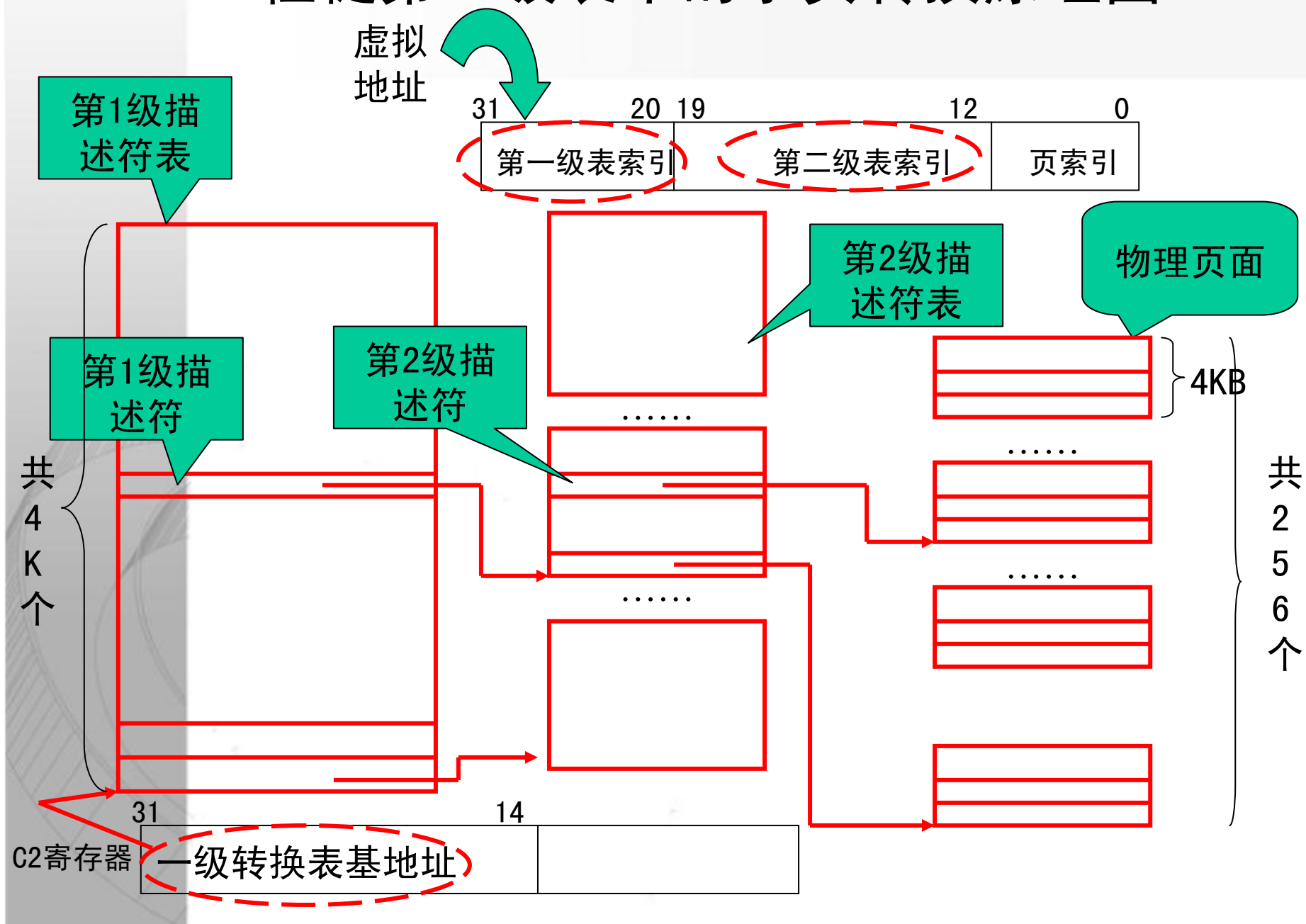
对于一个第二级描述符，有四种可能，由描述符的bits[1:0]选择。

- bits[1:0]==0b00，说关联的虚拟地址没有被映射
- bits[1:0]==0b01，这个入口是大页描述符，描述64KB 的虚拟地址。
- bits[1:0]== 0b10，这个入口是小页描述符，描述4KB 的虚拟地址。
- bits[1:0]== 0b11，这个入口是微页描述符，描述1KB 的虚拟地址。

粗糙第二级表中的小页转换原理图



粗糙第二级表中的小页转换原理图



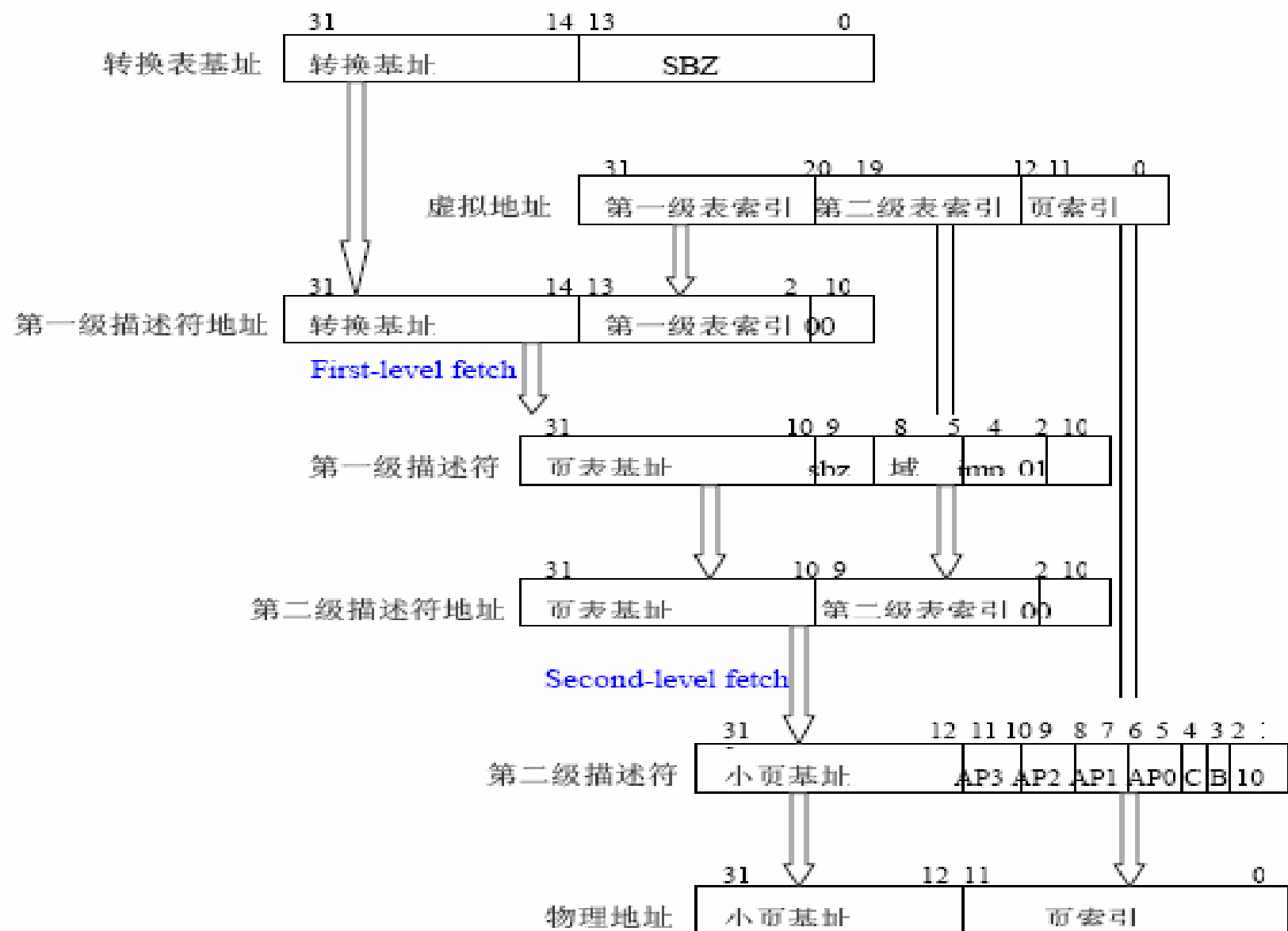


图 3-7 粗糙第二级表中的小页转换

ARM和X86处理器在内存映射方面异同

- ❖ X86处理器采用两层页面映射（段页式）
- ❖ 在ARM处理器上，如果整个段都有映射，就采用单层映射；如果不是整个段的映射，则采用两层映射

重要提示

- ❖ 在ARM处理器上，如果整个段都有映射，就采用单层映射；
- ❖ 如果不是整个段的映射，则采用两层映射

X86下Linux内存管理

内存管理

❖ 概述

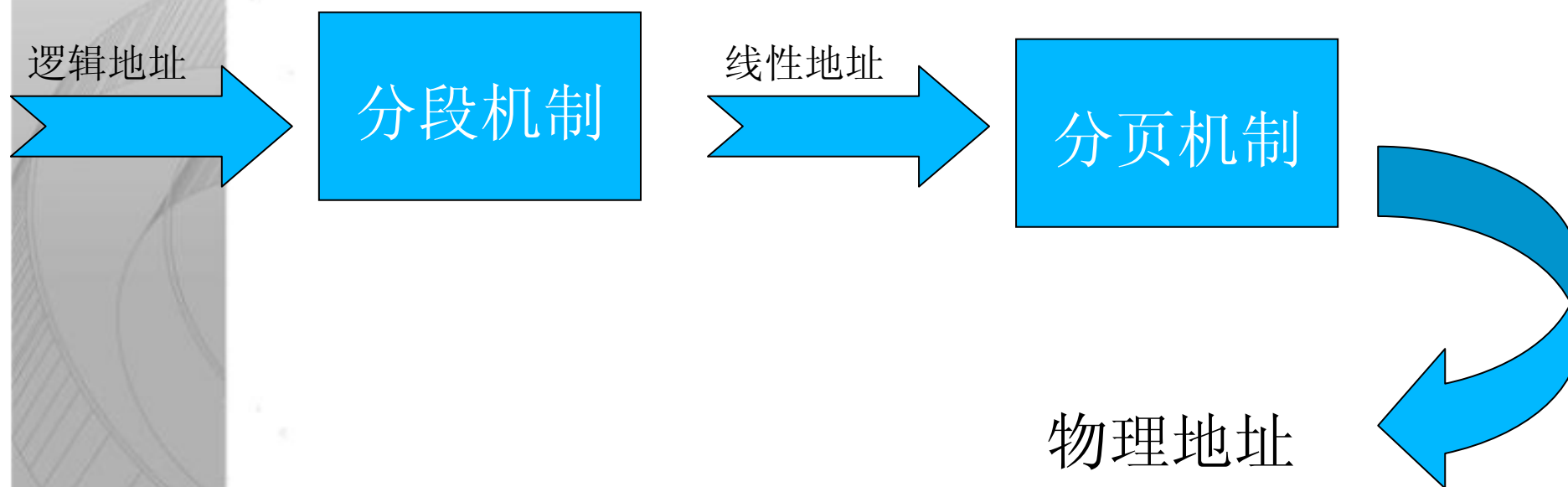
❖ Linux支持很多硬件运行平台，常用的有：Intel X86，Alpha，Sparc，ARM等。对于不能够通用的一些功能，Linux必须依据硬件平台的特点来具体实现。

❖ x86 内存架构

- 逻辑地址 (**logical address**) 是存储位置的地址，它可能直接对应于一个物理位置，也可能不直接对应于一个物理位置。逻辑地址通常在请求控制器中的信息时使用。
- 线性地址 (**linear address**) (或称为 平面地址空间) 是从 **0** 开始进行寻址的内存。之后的每个字节都可顺序使用下一数字来引用 (**0、1、2、3** 等)，直到内存末尾为止。这就是大部分非 **Intel CPU** 的寻址方式。**Intel?** 架构使用了分段的地址空间，其中内存被划分成 **64KB** 的段，有一个段寄存器总是指向当前正在寻址的段的基址。这种架构中的 **32** 位模式被视为平面地址空间，不过它也使用了段。
- 物理地址 (**physical address**) 是使用物理地址总线中的位表示的地址。物理地址可能与逻辑地址不同，内存管理单元可以将逻辑地址转换成物理地址。

(续)

- ❖ CPU 使用两种单元将逻辑地址转换成物理地址。第一种称为分段单元 (segmented unit)，另外一种称为分页单元 (paging unit)。



分段使用到的数据结构

- ❖ 全局描述符表（GDT，Global Descriptor Table）：存放系统用的段描述符和各项任务共用的段描述符，可以是上述的任何一类段的段描述符，最大表长64KB
- ❖ 局部描述符表（LDT，Local Descriptor Table）：存放某个任务专用的各段的段描述符，只能是三类进程段的段描述符和调用门描述符，最大表长4GB
- ❖ 段描述符（Segment Descriptor）：64bits，用来描述一个段的基地址（该地址是线性地址），该段的类型，对该段操作的限制
- ❖ 门描述符（Gate Descriptor）：64bits，一种特殊的描述符，为处于不同特权级的系统调用或程序的调用或访问提供保护；分为四类：调用门描述符（Call Gate Descriptor）、中断门描述符（Interrupt Gate Descriptor）、陷阱门描述符（Trap Gate

分段使用到的数据结构（续）

- ❖ 段选择符（Segment Selector）：16bits，用于在GDT或LDT中索引相应的段描述符
- ❖ 中断描述表（IDT，Interrupt Describer Table）：存放门描述符，只能是中断门描述符，陷阱门描述符和任务门描述符，最大表长64KB

分段使用到的寄存器

- ❖ 全局描述符表寄存器（GDTR，GDT Register）：
48bits，32bits为GDT的基地址（线性地址），16bits为GDT的表长；GDTR的初始值为：基地址0，表长0xFFFF；
- ❖ 局部描述符表寄存器（LDTR，LDT Register）：
80bits，16bits为LDT段选择符，64bits为该LDT段的段描述符；
- ❖ 中断描述符表寄存器（IDTR，IDT Register）：
48bits，32bits为IDT的基地址（线性地址），16bits为IDT的表长；IDTR的初始值为：基地址0，表长0xFFFF；
- ❖ 任务寄存器（TR，Task Register）：80bits，16bits为任务状态段选择符，64bits为该任务状态段的段描述符；
- ❖ 六个段寄存器（Segment Register）：分为可见部分和隐藏部分，可见部分为段选择符，隐藏部分为段描述符；
这六个段寄存器分别为CS、SS、DS、ES、FS、GS；关于这些段寄存器的作用参见[1]中3.4.2 'Segment Register'；

4.1 段控制单元模型概述

□ 段由两个元素构成

- 基址 (**base address**) 包含某个物理内存位置的地址
- 长度值 (**length value**) 指定该段的长度

□ 分段地址还包括两个组件

- 段选择器 (**segment selector**) --指定了要使用的段 (即基址和长度值)
- 段内偏移量 (**offset into the segment**)--指定了实际内存位置相对于基址的偏移量

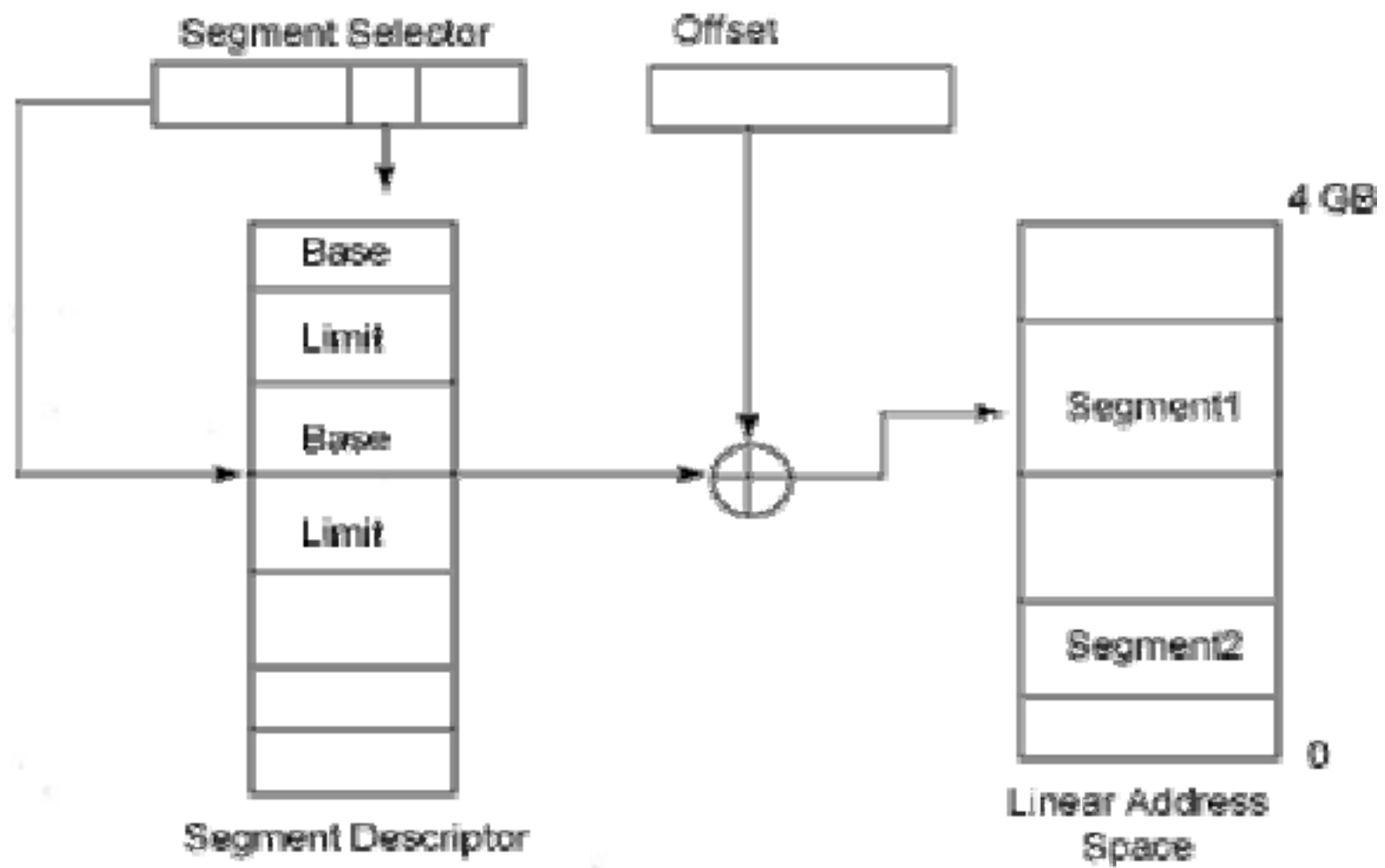
段描述符和段寄存器的相互关系

□ 段选择器包含以下内容：

- 一个 13 位的索引，用来标识 GDT 或 LDT 中包含的对应段描述符条目
- TI (Table Indicator) 标志指定段描述符是在 GDT 中还是在 LDT 中，如果该值是 0，段描述符就在 GDT 中；如果该值是 1，段描述符就在 LDT 中。
- RPL (request privilege level) 定义了在为对应的段选择器加载到段寄存器中时 CPU 的当前特权级别。

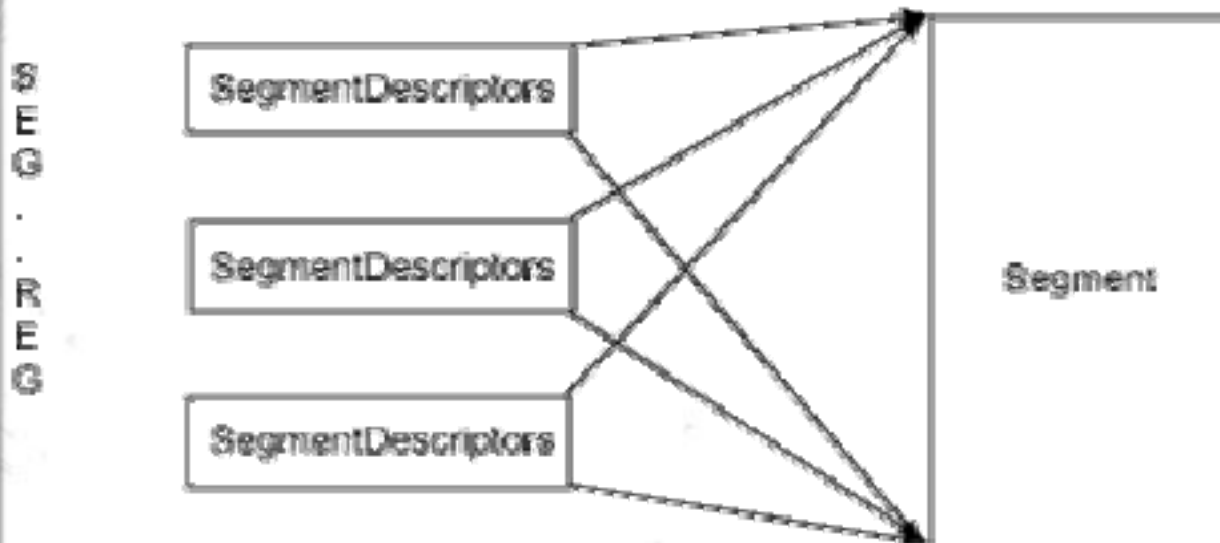
□ 每次将段选择器加载到段寄存器中时，对应的段描述符都会从内存加载到相匹配的不可编程 CPU 寄存器中。每个段描述符长 8 个字节，表示内存中的一个段。这些都存储到 LDT 或 GDT 中。段描述符条目中包含一个指针和一个 20 位的值（Limit 字段），前者指向由 Base 字段表示的相关段中的第一个字节，后者表示内存中段的大小。

逻辑地址获得线性地址



Linux对分段机制的支持

- ❑ 在 Linux 中，所有的段寄存器都指向相同的段地址范围 —— 换言之，每个段寄存器都使用相同的线性地址
- ❑ 优点
 - 当所有的进程都使用相同的段寄存器值时（当它们共享相同的线性地址空间时），内存管理更为简单。
 - 在大部分架构上都可以实现可移植性。某些 RISC 处理器也可通过这种受限的方式支持分段。



GDT 中的内核代码段描述符

- ❑ Base = 0x00000000
- ❑ Limit = 0xffffffff ($2^{32} - 1$) = 4GB
- ❑ G (粒度标志) = 1, 表示段的大小是以页为单位表示的
- ❑ S = 1, 表示普通代码或数据段
- ❑ Type = 0xa, 表示可以读取或执行的代码段
- ❑ DPL 值 = 0, 表示内核模式
- ❑ 选择器存储在 cs 寄存器中
- ❑ Linux 中用来访问这个段选择器的宏是 `_KERNEL_CS`

内核数据段 (*kernel data segment*) 描述符

- 与内核代码段的值类似，惟一不同的就是 **Type** 字段值为 2。这表示此段为数据段，选择器存储在 ds 寄存器中。**Linux** 中用来访问这个段选择器的宏是 `_KERNEL_DS`。

用户代码段 (*user code segment*)

- ❑ Base = 0x00000000
- ❑ Limit = 0xffffffff
- ❑ G = 1
- ❑ S = 1
- ❑ Type = 0xa，表示可以读取和执行的代码段
- ❑ DPL = 3，表示用户模式
- ❑ 在 Linux 中，我们可以通过 `_USER_CS` 宏来访问此段选择器

用户数据段 (*user data segment*) 描述符

- 和用户代码段类似，惟一不同的字段就是 **Type**，它被设置为 2，表示将此数据段定义为可读取和写入。Linux 中用来访问此段选择器的宏是 `_USER_DS`。

TSS 段 (TSS segment) 描述符

- 每个 TSS 段 (TSS segment) 描述符都代表一个不同的进程。TSS 中保存了每个 CPU 的硬件上下文信息，它有助于有效地切换上下文
- 每个进程都有自己在 GDT 中存储的对应进程的 TSS 描述符。这些描述符的值如下
 - **Base = &tss** (对应进程描述符的 TSS 字段的地址；例如 `&tss_struct`) 这是在 Linux 内核的 `schedule.h` 文件中定义的
 - **Limit = 0xeb** (TSS 段的大小是 236 字节)
 - **Type = 9 或 11**
 - **DPL = 0**。用户模式不能访问 TSS。G 标志被清除

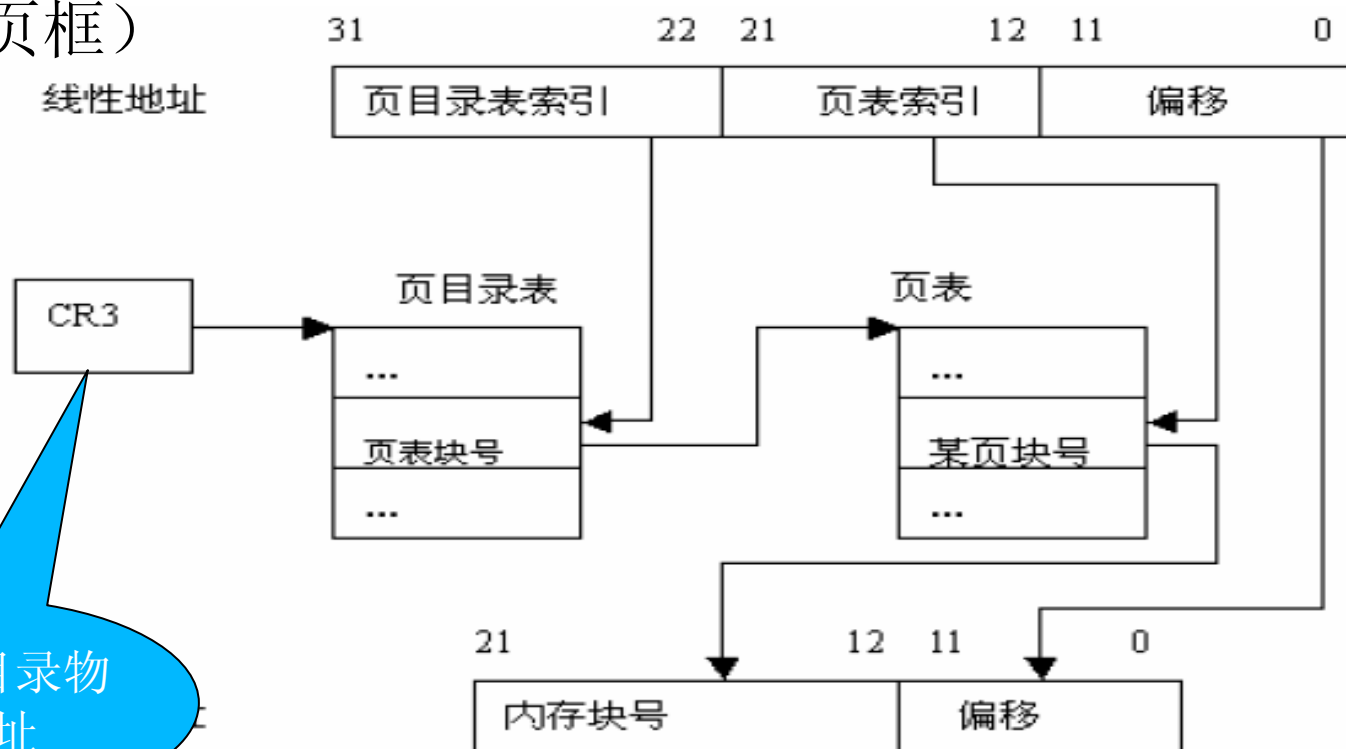
4.2 分页模型概述

- 分页单元负责将线性地址转换成物理地址
- 线性地址会被分组成页的形式，这些线性地址实际上都是连续的——分页单元将这些连续的内存映射成对应的连续物理地址范围（称为 页框）



X86分页机制

❖ 线性地址到对应物理位置的转换的过程包含两个步骤。第一步使用了一个称为 **页目录 (Page Directory)** 的转换表（从页目录转换成页表），第二步使用了一个称为 **页表 (Page Table)** 的转换表（即页表加偏移量再加页框）



存放页目录物理地址

Linux分页机制

- ❖ Linux 采用的是一个体系结构无关的三级页表模型

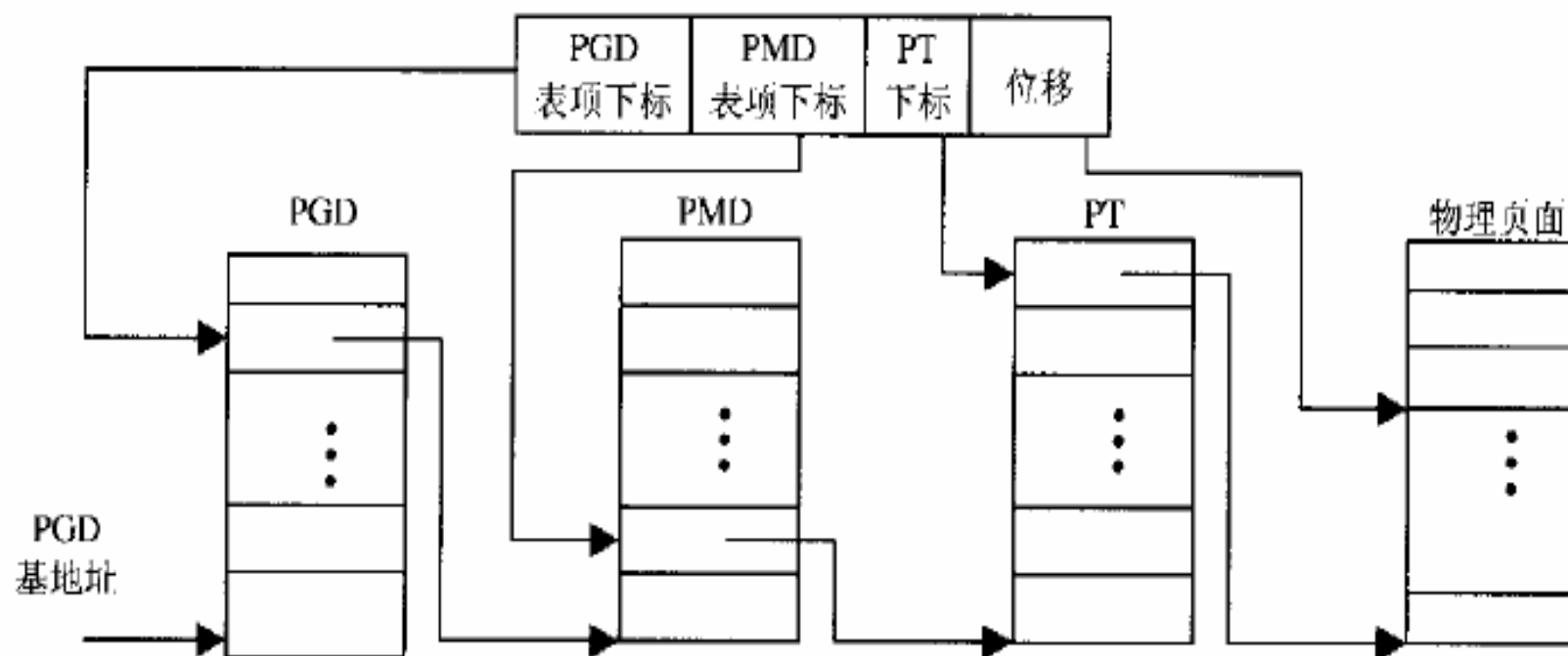


图 2.1 三层地址映射示意图

例：

- `//aa.cpp`
- `#include <stdio.h>`
- `Void printhello()`
- `{`
- `printf("hello,world!\n");`
- `}`
- `Void main()`
- `{`
- `printhello();`
- `}`
- 编译：`gcc a.cpp -o aa`

- objdump -d aa

08048328 <printhello>:

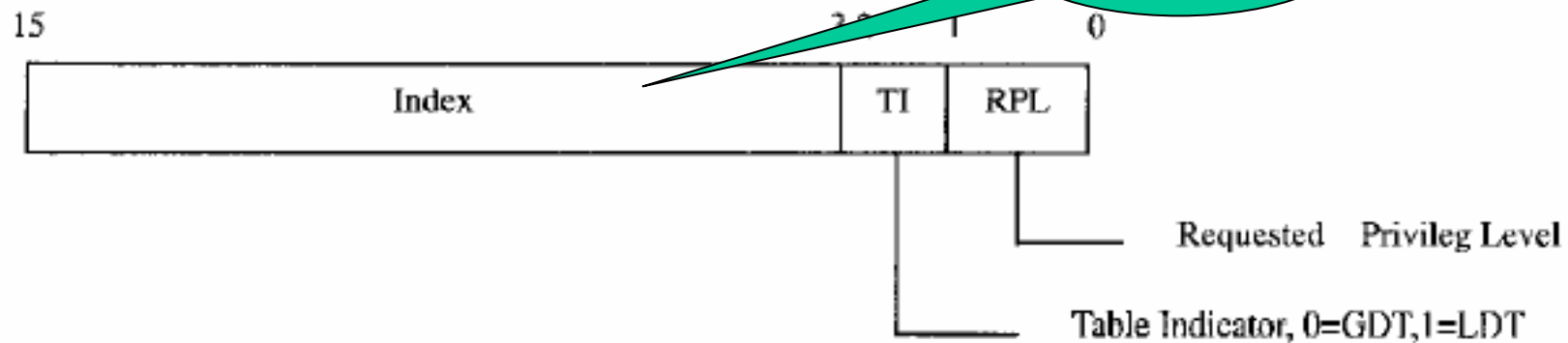
| | | | |
|----------|----------------|-------|----------------------|
| 8048328: | 55 | push | %ebp |
| 8048329: | 89 e5 | mov | %esp,%ebp |
| 804832b: | 83 ec 08 | sub | \$0x8,%esp |
| 804832e: | 83 ec 0c | sub | \$0xc,%esp |
| 8048331: | 68 04 84 04 08 | push | \$0x8048404 |
| 8048336: | e8 2d ff ff ff | call | 8048268 <_init+0x38> |
| 804833b: | 83 c4 10 | add | \$0x10,%esp |
| 804833e: | c9 | leave | |
| 804833f: | c3 | ret | |

08048340 <main>:

| | | | |
|----------|----------------|-------|----------------------|
| 8048340: | 55 | push | %ebp |
| 8048341: | 89 e5 | mov | %esp,%ebp |
| 8048343: | 83 ec 08 | sub | \$0x8,%esp |
| 8048346: | 83 e4 f0 | and | \$0xfffffffff0,%esp |
| 8048349: | b8 00 00 00 00 | mov | \$0x0,%eax |
| 804834e: | 29 c4 | sub | %eax,%esp |
| 8048350: | e8 d3 ff ff ff | call | 8048328 <printhello> |
| 8048355: | c9 | leave | |
| 8048356: | c3 | ret | |
| 8048357: | 90 | nop | |

⌋

第一步——一段式映射



- ❖ 内核在建立进程时就会调用下列代码将CS的值设定好
- ❖ (include/asm-i386/processor.h)
- ❖ #define start_thread(regs, new_eip, new_esp) do { \
- ❖ __asm__("movl %0,%%fs ; movl %0,%%gs": : "r" (0)); \
- ❖ set_fs(USER_DS); \
- ❖ regs->xds = __USER_DS; \
- ❖ regs->xes = __USER_DS; \
- ❖ regs->xss = __USER_DS; \
- ❖ regs->xcs = __USER_CS; \
- ❖ regs->eip = new_eip; \
- ❖ regs->esp = new_esp; \
- ❖ } while (0)

再来看看 USER_CS 和 USER_DS 到底是什么。那是在 include/asm-i386/segment.h 中定义的：

```
4  #define __KERNEL_CS      0x10
5  #define __KERNEL_DS      0x18
6
7  #define __USER_CS        0x23
8  #define __USER_DS        0x2B
```

也就是说，Linux 内核中只使用四种不同的段寄存器数值，两种用于内核本身，两种用于所有的进程。现在，我们将这四种数值用二进制展开并与段寄存器的格式相对照：

| | | Index | | | | | | | | TI | RPL | | |
|-------------|------|-------|---|---|---|---|---|---|---|----|-----|---|---|
| __KERNEL_CS | 0x10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| __KERNEL_DS | 0x18 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| __USER_CS | 0x23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| __USER_DS | 0x2B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

一对照就清楚了，那就是：

| | | | |
|--------------|------------|---------|---------|
| __KERNEL_CS: | index = 2, | TI = 0, | RPL = 0 |
| __KERNEL_DS: | index = 3, | TI = 0, | RPL = 0 |
| __USER_CS: | index = 4, | TI = 0, | RPL = 3 |
| __USER_DS: | index = 5, | TI = 0, | RPL = 3 |

回到我们的程序中。我们的程序显然不属于内核，所以在进程的用户空间中运行，内核在调度该进程进入运行时，把 CS 设置成 `__USER_CS`，即 0x23。所以，CPU 以 4 为下标，从全局段描述表 GDT 中找对应的段描述项。

初始的 GDT 内容是在 `arch/i386/kernel/head.S` 中定义的，其主要内容在运行中并不改变：

```
444  /*
445   * This contains typically 140 quadwords, depending on NR_CPUS.
446   *
447   * NOTE! Make sure the gdt descriptor in head.S matches this if you
448   * change anything.
449   */
450  ENTRY(gdt_table)
451      .quad 0x0000000000000000    /* NULL descriptor */
452      .quad 0x0000000000000000    /* not used */
453      .quad 0x00cf9a000000ffff    /* 0x10 kernel 4GB code at 0x00000000 */
454      .quad 0x00cf92000000ffff    /* 0x18 kernel 4GB data at 0x00000000 */
455      .quad 0x00cffa000000ffff    /* 0x23 user   4GB code at 0x00000000 */
```

```

K_CS: 0000 0000 1100 1111 1001 1010 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
K_DS: 0000 0000 1100 1111 1001 0010 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
K_CS: 0000 0000 1100 1111 1111 1010 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
K_DS: 0000 0000 1100 1111 1111 0010 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111

```

读者结合下页图 2.4 段描述项的定义仔细对照，可以得出如下结论：

(1) 四个段描述项的下列内容都是相同的。

- B0-B15、B16-B31 都是 0 ——基地址全为 0；
- L0-L15、L16-L19 都是 1 ——段的上限全是 0xffff；
- G 位都是 1 ——段长单位均为 4KB；
- D 位都是 1 ——对四个段的访问都是 32 位指令；
- P 位都是 1 ——四个段都在内存。

结论：每个段都是从 0 地址开始的整个 4GB 虚存空间，虚地址到线性地址的映射保持原值不变。

因此，讨论或理解 Linux 内核的页式映射时，可以直接将线性地址当作虚拟地址，二者完全一致。

(2) 有区别的地方只是在 bit40~bit 46，对应于描述项中的 type 以及 S 标志和 DPL 位段。

- 对 KERNEL_CS： DPL=0，表示 0 级；S 位为 1，表示代码段或数据段；type 为 1010，表示代码段，可读，可执行，尚未受到访问。
- 对 KERNEL_DS： DPL=0，表示 0 级；S 位为 1，表示代码段或数据段；type 为 0010，表示数据段，可读，可写，尚未受到访问。
- 对 USER_CS： DPL=3，表示 3 级；S 位为 1，表示代码段或数据段；type 为 1010，表示代码段，可读，可执行，尚未受到访问。
- 对 USER_DS：即下标为 5 时，DPL=3，表示 3 级；S 位为 1，表示代码段或数据段；type 为 0010，表示数据段，可读，可写，尚未受到访问。

- 因此0X8048328这个虚拟地址经过段式映射后得到线性地址，其值仍然为0X8048328
- 下面进入重要的页式映射阶段

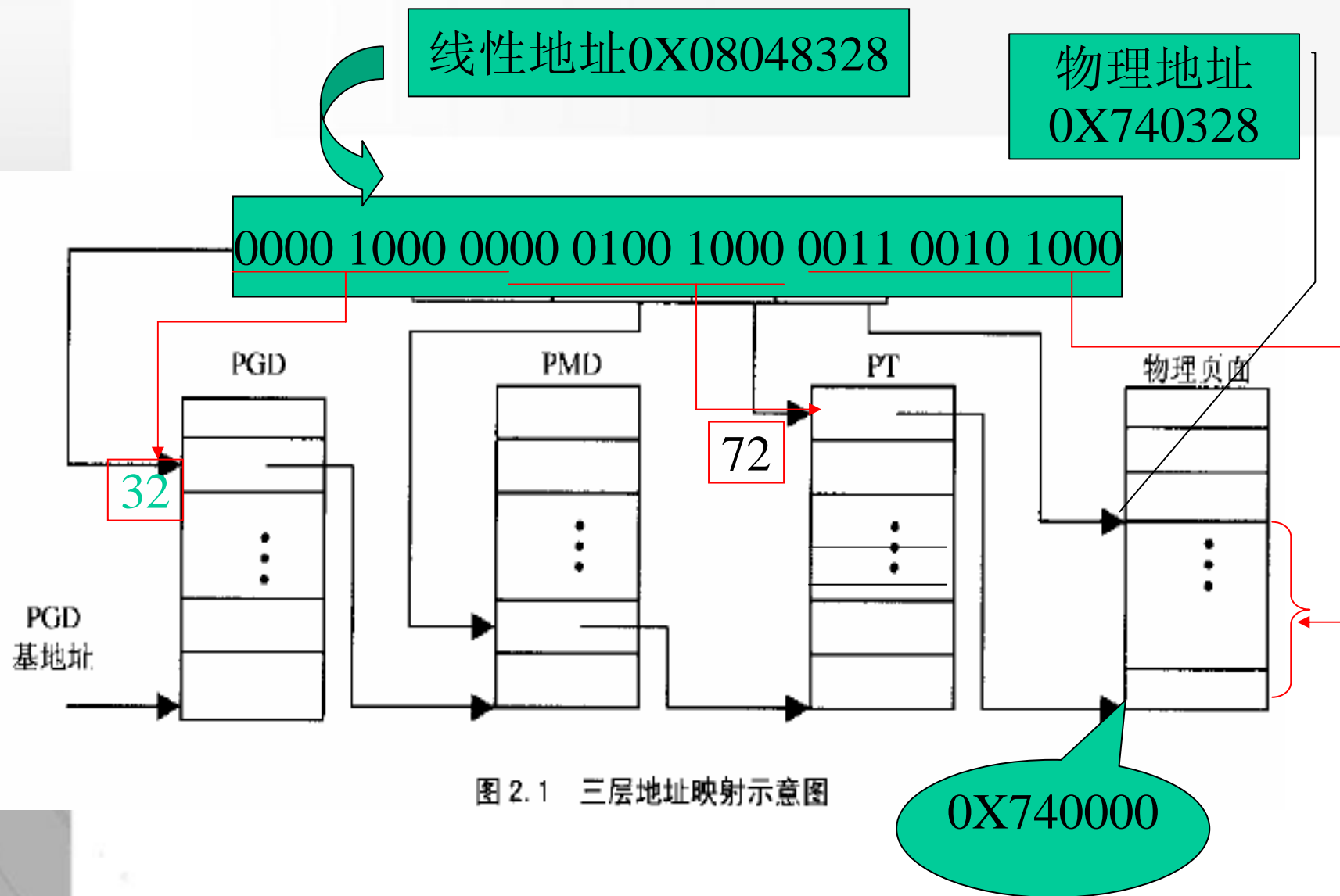


图 2.1 三层地址映射示意图

Linux进程管理

进程基本概念

- 在命令终端下敲入：
#ps -A

```
root@localhost:~  
文件(E) 编辑(E) 查看(V) 终端(T) 转到(G) 帮助(H)  
[root@localhost root]#  
[root@localhost root]# ps -A  
  PID TTY          TIME CMD  
    1 ?           00:00:08 init  
    2 ?           00:00:02 keventd  
    3 ?           00:00:01 kpmnd  
    4 ?           00:00:00 ksoftirqd_CPU0  
    9 ?           00:00:09 bdfush  
    5 ?           00:00:17 kswapd  
    6 ?           00:00:13 kscand/DMA  
    7 ?           00:00:44 kscand/Normal  
    8 ?           00:00:00 kscand/HighMem  
   10 ?           00:00:03 kupdated  
   11 ?           00:00:00 nrecoveryd  
   19 ?           00:00:36 kjournald  
   77 ?           00:00:00 khubd  
  1167 ?          00:00:00 kjournald  
  1463 ?          00:00:01 syslogd  
  1467 ?          00:00:00 klogd  
  1477 ?          00:00:00 portmap  
  1496 ?          00:00:00 rpc.statd  
  1575 ?          00:03:44 vmware-guestd  
  1631 ?          00:00:00 apmd  
  1668 ?          00:00:02 sshd  
  1682 ?          00:00:00 xinetd  
  1696 ?          00:00:00 rpc.rquotad  
  1700 ?          00:00:00 nfsd  
  1701 ?          00:00:00 nfsd
```

进程和程序的区别

- 通俗的讲程序是一个包含可以执行代码的文件,是一个静态的文件,这个静态的文件存放在磁盘等介质中。一旦这个静态的文件被我们执行起来,也就是说加载到内存中去,此时在内存中就创建了该程序的一个动态执行实例
- 当程序被Linux系统调用到内存以后, Linux系统会给程序分配一定的资源(内存,设备等等)然后进行一系列的复杂操作,使程序变成进程以供系统调用。
- 为了区分各个不同的进程,系统给每一个进程分配了一个ID(就象我们的身份证)以便识别。

进程的定义

- (1) 进程是一个抽象实体，当它执行某个任务时，系统会为其分配各种资源，当它结束时，系统会收回这些资源。
- (2) 进程是可以并行执行的计算部分。
- (3) 进程是独立的可被调度的一次程序实践活动。

Linux下进程的描述

- 进程标识符：当一个进程产生时，系统都会为它分配一个标识符
- 进程所占的内存区域：每个进程执行时都需要占用一定的内存区域，此区域用于保存该进程所运行的程序代码和使用的程序变量。每一个进程所占用的内存是相互独立的，因此改变一个进程所占内存中数据的任何改动，都只对该进程产生影响，不会影响到其它进程的顺利执行
- 文件描述符：当一个进程在执行时，它需要使用一些相关的文件描述符。文件描述符描述了被打开文件的信息，不同的进程打开同一个文件时，所使用的文件描述符是不同的。一个进程文件描述符的改变并不会对其它的进程打开同一个文件的描述符产生任何影响。

Linux下进程描述（续）

- 安全信息：一个进程的安全信息包括用户识别号和组识别号
- 进程环境：一个进程的运行环境包括环境变量和启动该进程的程序调用的命令行。
- 信号处理：一个进程有时需要用信号同其它进程进行通信。进程可以发送和接收信号，并对其作出相应处理。
- 资源安排：进程是调度系统资源的基本单位。当多个进程同时运行时，linux系统内核安排不同进程轮流使用系统的各种资源。

Linux 下进程描述

- 同步处理：多个程序之间同步运行的实现，也是通过进程来完成的。这将会使用到诸如共享内存、文件锁定等方法。
- 进程状态：在一个进程存在期间，每一时刻进程都处在一定的状态，包括运行、等待被调度或睡眠状态。

Linux下进程结构体描述

- 为了更好的描述进程，在Linux系统中专门定义了一个新的结构体，叫做struct task_struct
- 该结构体定义文件位于include/linux/sched.h

Linux下进程结构体描述（续）

- ❖ 内核程序通过进程表对进程进行管理，每个进程在进程表中占有一项
- ❖ 在Linux 系统中，进程表项是一个task_struct 任务结构指针（进程控制块PCB（Process Control Block）或进程描述符PD（Processor Descriptor））
- ❖ 任务数据结构定义在头文件include/linux/sched.h 中
- ❖ task_struct 任务结构包括进程当前运行的状态信息、信号、进程号、父进程号、运行时间累计值、正在使用的文件和本任务的局部描述符以及任务状态段信息

```

struct task_struct {
    long state           //任务的运行状态 (-1 不可运行, 0 可运行(就绪), >0 已停止)。
    long counter         // 任务运行时间计数(递减)(滴答数), 运行时间片。
    long priority        // 运行优先数。任务开始运行时 counter=priority, 越大运行越长。
    long signal          // 信号。是位图, 每个比特位代表一种信号, 信号值=位偏移值+1。
    struct sigaction sigaction[32] // 信号执行属性结构, 对应信号将要执行的操作和标志信息。
    long blocked         // 进程信号屏蔽码(对应信号位图)。
    int exit_code        // 任务执行停止的退出码, 其父进程会取。
    unsigned long start_code // 代码段地址。
    unsigned long end_code  // 代码长度(字节数)。
    unsigned long end_data  // 代码长度 + 数据长度(字节数)。
    unsigned long brk       // 总长度(字节数)。
    unsigned long start_stack // 堆栈段地址。
    long pid              // 进程标识号(进程号)。
    long father           // 父进程号。
    long pgrp             // 父进程组号。
    long session          // 会话号。
    long leader           // 会话首领。
    unsigned short uid     // 用户标识号(用户 id)。
    unsigned short euid    // 有效用户 id。
    unsigned short suid    // 保存的用户 id。
    unsigned short gid     // 组标识号(组 id)。
    unsigned short egid    // 有效组 id。
    unsigned short sgid    // 保存的组 id。
    long alarm             // 报警定时值(滴答数)。
    long utime             // 用户态运行时间(滴答数)。
    long stime             // 系统态运行时间(滴答数)。
    long cutime            // 子进程用户态运行时间。
    long cstime            // 子进程系统态运行时间。
    long start_time        // 进程开始运行时刻。
    unsigned short used_math // 标志: 是否使用了协处理器。

```

task_struct结构体解读

一进程标识

- 进程的标识：进程号、父进程号、父进程组号、会话号、会话首领号都是用于从不同的侧面来描述进程的身份

task_struct结构体解读

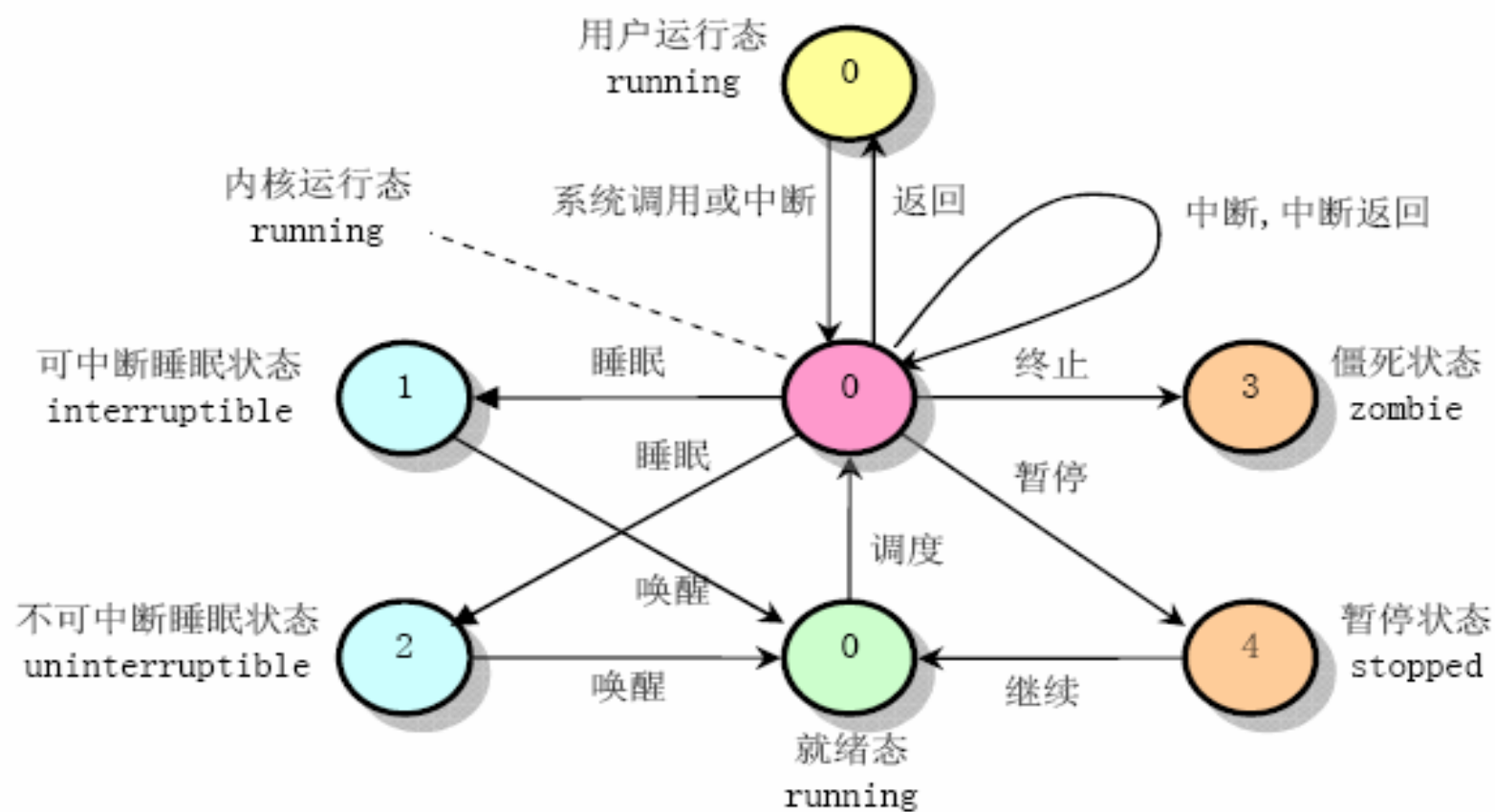
一 进程状态

- ❖ 一个进程在其生存期内，可处于一组不同的状态下，称为进程状态
- ❖ 运行状态（**TASK_RUNNING**）—当进程正在被CPU 执行，或已经准备就绪随时可由调度程序执行，则称该进程为处于运行状态（**running**）。进程可以在内核态运行，也可以在用户态运行
- ❖ 可中断睡眠状态（**TASK_INTERRUPTIBLE**）—当进程处于可中断等待状态时，系统不会调度该进程执行。当系统产生一个中断或者释放了进程正在等待的资源，或者进程收到一个信号，都可以唤醒进程转换到就绪状态（运行状态）。

(续)

- ❖ 不可中断睡眠状态 (**TASK_UNINTERRUPTIBLE**)
— 与可中断睡眠状态类似。但处于该状态的进程只有被使用 **wake_up()** 函数明确唤醒时才能转换到可运行的就绪状态。
- ❖ 暂停状态 (**TASK_STOPPED**) — 当进程收到信号 **SIGSTOP**、**SIGTSTP**、**SIGTTIN** 或 **SIGTTOU** 时就会进入暂停状态。可向其发送 **SIGCONT** 信号让进程转换到可运行状态。
- ❖ 僵死状态 (**TASK_ZOMBIE**) — 当进程已停止运行，但其父进程还没有询问其状态时，则称该进程处于僵死状态。

(续)

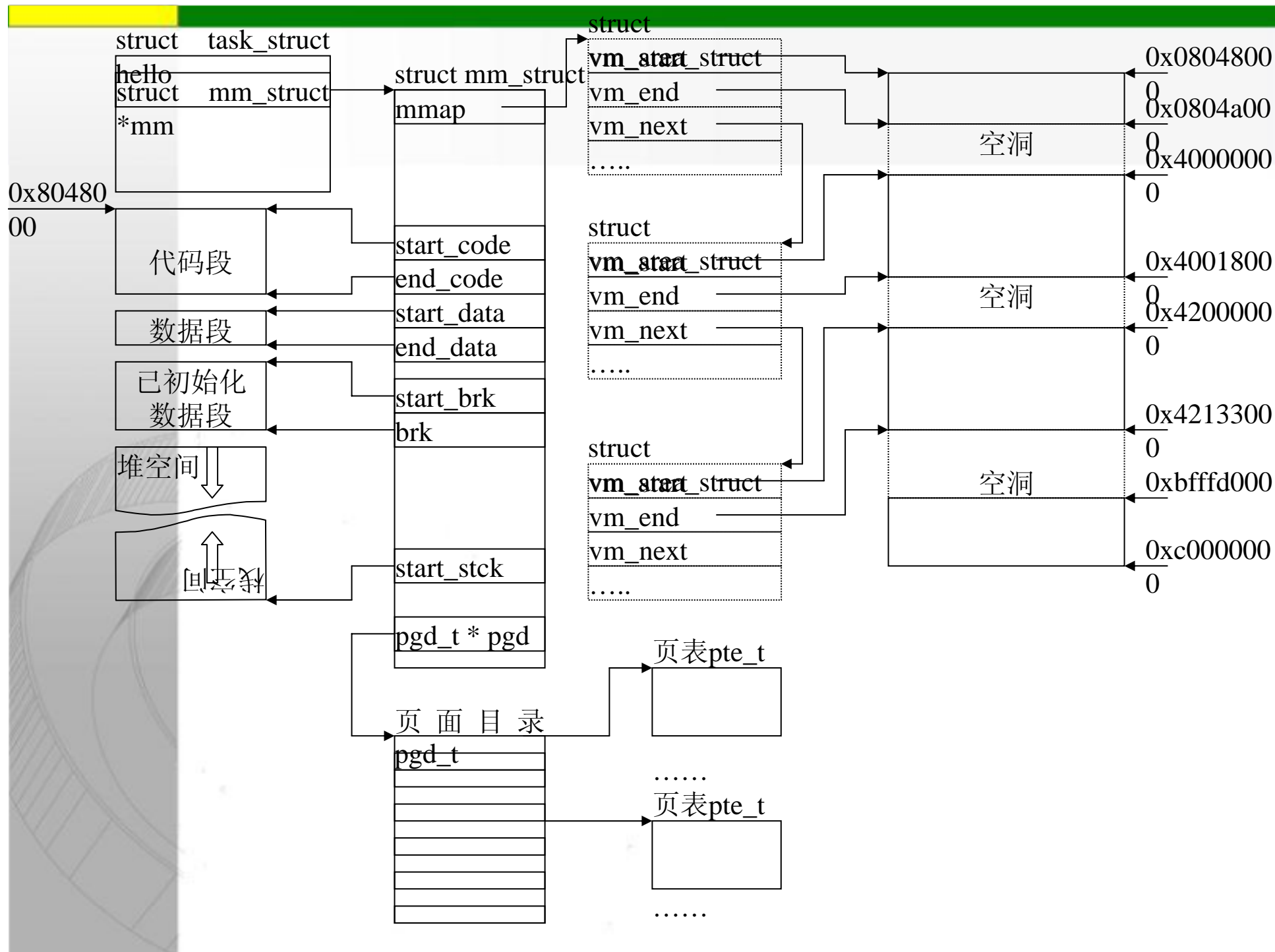


进程状态及转换关系

task_struct结构体解读

— 内存

- 虚拟内存管理
- 物理内存管理



task_struct结构体解读

一 文件系统

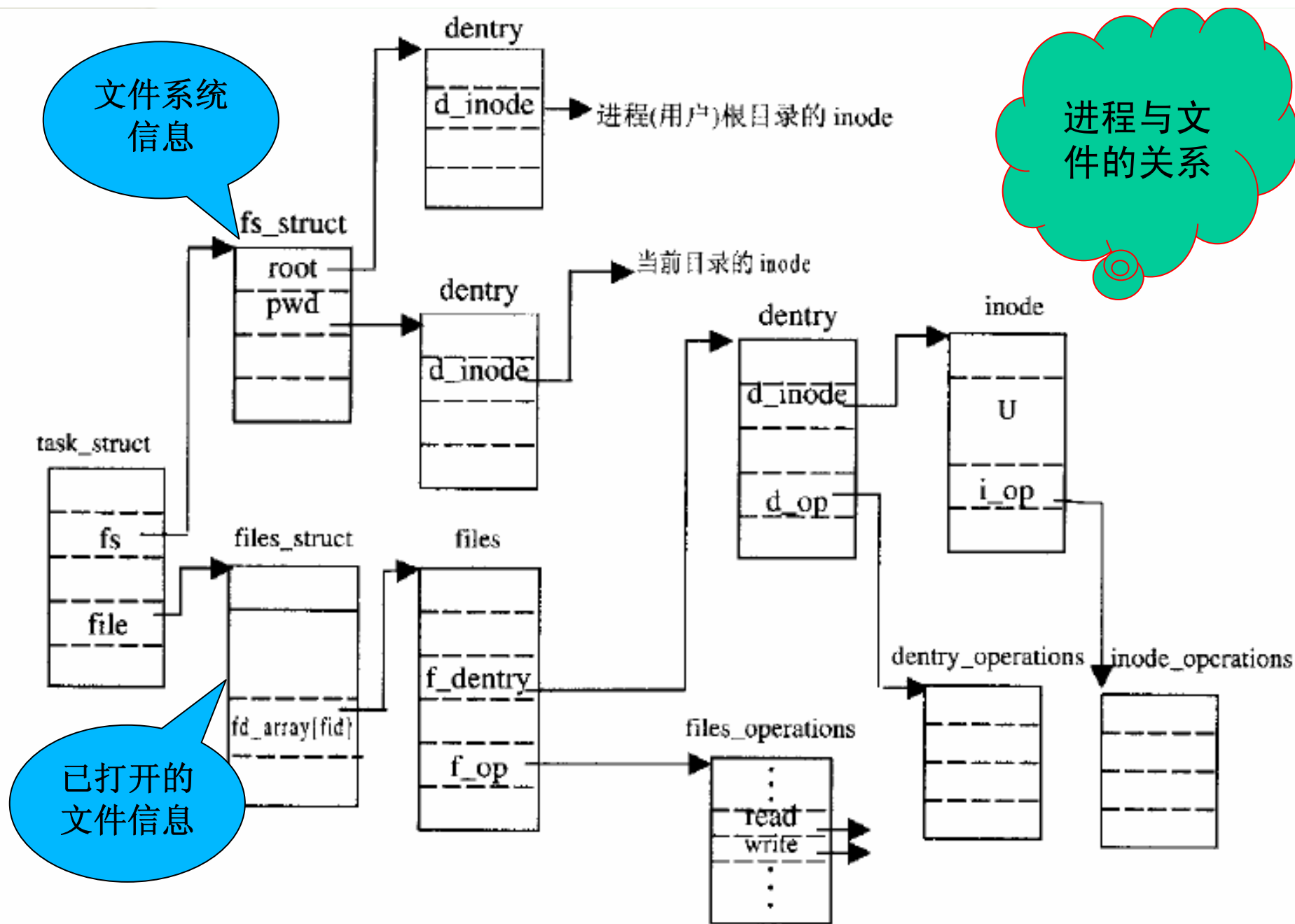


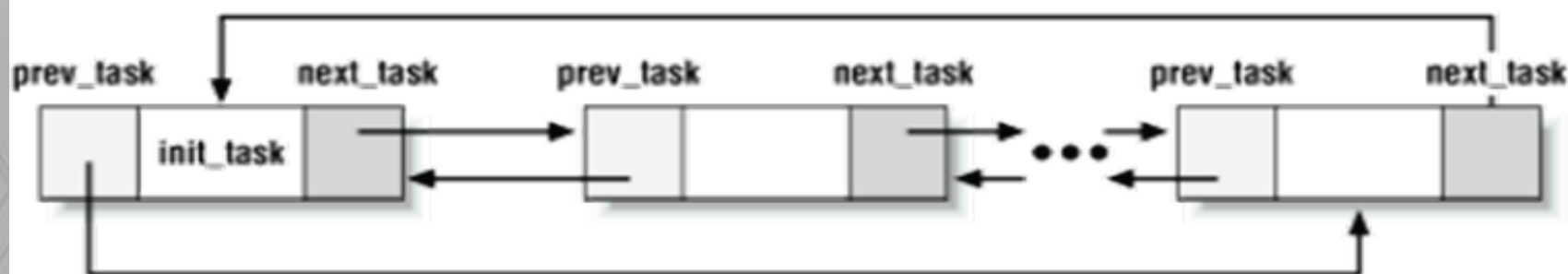
图 5.2 Linux 文件系统逻辑结构图

进程组织

- 进程间的父子关系（续）
- Linux中的0号进程,通常称为swapper进程，是所有进程的祖先。由它执行cpu_idle()函数，当没有其他进程处于TASK_RUNNING的时候，调度程序会选择0号进程运行
- 0号进程创建1号进程，通常称为init进程。它创建和监控其他进程的活动

进程组织

- **task[]**数组（实际是双向链表指针）
 - 包含指向系统中所有**task_struct**结构的指针
 - 数组大小限制了系统中的进程数目
 - 将所有任务串连起来



进程调度

- ❖ Linux 进程是抢占式的。被抢占的进程仍然处于 `TASK_RUNNING` 状态，只是暂时没有被CPU运行。进程的抢占发生在进程处于用户态执行阶段，在内核态执行时是不能被抢占的。
- ❖ 在Linux 0.11 中采用了基于优先级排队的调度策略

调度方式

- Linux中的每个进程都分配有一个相对独立的虚拟地址空间。该虚存空间分为两部分：用户空间包含了进程本身的代码和数据；内核空间包含了操作系统的代码和数据。
- Linux采用“有条件的可剥夺”调度方式。对于普通进程，当其时间片结束时，调度程序挑选出下一个处于TASK_RUNNING状态的进程作为当前进程（自愿调度）。对于实时进程，若其优先级足够高，则会从当前的运行进程中抢占CPU成为新的当前进程（强制调度）。发生强制调度时，若进程在用户空间中运行，就会直接被剥夺CPU；若进程在内核空间中运行，即使迫切需要其放弃CPU，也仍要等到从它系统空间返回的前夕才被剥夺CPU。

调度策略

- **SCHED_OTHER:** SCHED_OTHER是面向普通进程的时间片轮转策略。采用该策略时，系统为处于TASK_RUNNING状态的每个进程分配一个时间片。当时间片用完时，进程调度程序再选择下一个优先级相对较高的进程，并授予CPU使用权。
- **SCHED_FIFO:** SCHED_FIFO策略适用于对响应时间要求比较高，运行所需时间比较短的实时进程。采用该策略时，各实时进程按其进入可运行队列的顺序依次获得CPU。除了因等待某个事件主动放弃CPU，或者出现优先级更高的进程而剥夺其CPU之外，该进程将一直占用CPU运行。
- **SCHED_RR:** SCHED_RR策略适用于对响应时间要求比较高，运行所需时间比较长的实时进程。采用该策略时，各实时进程按时间片轮流使用CPU。当一个运行进程的时间片用完后，进程调度程序停止其运行并将其置于可运行队列的末尾。

task_struct结构体

—有关调度的成员

- Struct task_struct {
.....
volatile long need_resched; /*是否需要重新调度*/
long counter; /*进程当前还拥有的时间片*/
long nice; /*普通进程的动态优先级，来自UNIX系统*/
unsigned long policy; /*进程调度策略*/
unsigned long rt_priority; /*实时进程的优先级*/
.....
};

权值计算

- 为保证实时进程优于普通进程，Linux采取加权处理法。在进程调度过程中，每次选取下一个运行进程时，调度程序首先给可运行队列中的每个进程赋予一个权值`weight`。普通进程的权值就是其`counter`和优先级`nice`的综合，而实时进程的权值是它的`rt_priority`的值加1000，确保实时进程的权值总能大于普通进程。调度程序检查可运行队列中所有进程的权值，选取权值最大者作为下一个运行进程，保证了实时进程优先于普通进程获得CPU。

权值计算函数

—goodness()

- static inline goodness (struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
{

}

进程调度

- Linux的进程调度由调度程序schedule()完成，通过对schedule()的分析能更好理解调度的过程。

进程创建

- 在Linux系统中，一个新的进程是由一个已经存在的进程“复制”出来的
- 复制出来的子进程有自己的task_struct结构和系统空间堆栈，但与其父进程共享其它所有的资源
- 当系统调用fork创建一个进程的时候，它直接调用函数do_fork。do_fork函数拷贝父进程的相关数据，如文件、信号量、内存等。
- 完成进程初始化后，调用wake_up_new_task将其唤醒，状态变为TASK_RUNNING，挂到就绪队列，返回子进程的pid。

进程创建（续）

- `sys_fork()`
 - | - `do_fork()`
 - | - `copy_process()`
 - | - `dup_task_struct()`
 - | - `wake_up_new_task()`

进程创建（续）

- fork系统调用
 - 由**fork**创建的新进程称为子进程
 - 该函数被调用一次，会返回两次。给子进程的返回值是**0**，给父进程的返回值是子进程的进程ID
 - 然后子进程和父进程继续执行**fork**之后的指令。
 - 子进程拥有父进程数据空间，堆和栈的拷贝，但是它们并不是共享这些存储空间。这里用到了“写时复制”技术

进程创建（续）

- 写时复制技术

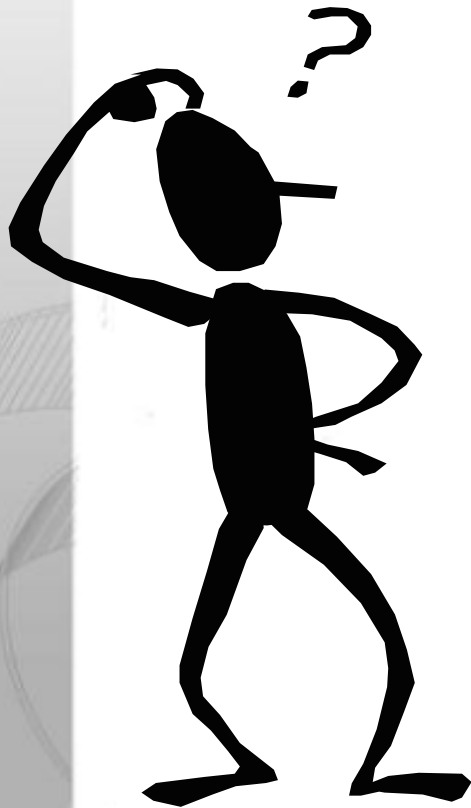
- 先通过复制页面表项暂时共享这个页面，将父、子进程可写虚拟内存页的页表项均标志为只读
- 当父或子进程向该内存页写入数据时，就会引起一次页面异常，页面异常处理程序将重新分配一个物理页面，并完成真正的内容复制
- 如前面的**fork**调用，逻辑拷贝整个进程的地址空间，仅当试图修改页面(产生写错误)才真正的拷贝

进程创建（续）

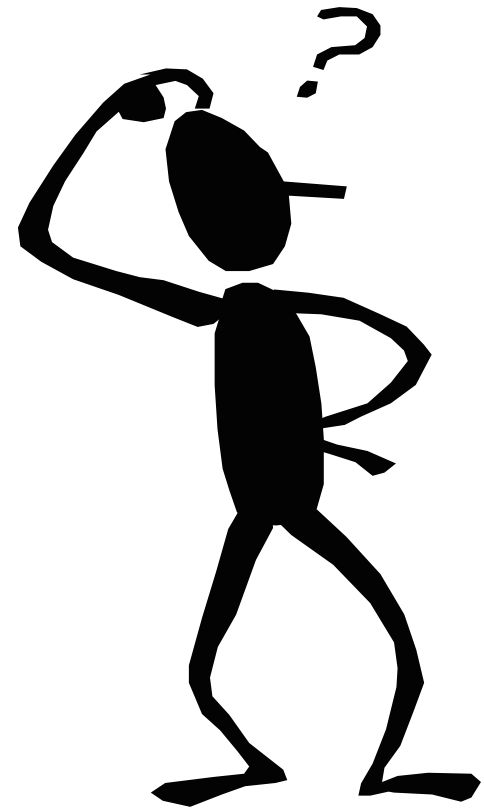
- 用fork创建子进程之后，为了让父进程和子进程执行不同的任务，经常需要调用一种exec函数以执行另一个程序。
- 当进程调用exec函数时，将先找到并打开给定的可执行程序文件，然后从文件中装入可执行程序并为其准备运行环境，之后便可装入并运行目标程序了。至此，该进程完全由新程序替代。新程序从main开始执行

进程创建（续）

- `exec`并不创建新进程，前后进程ID是不变的。
- 如果`exec`成功，调用者进程将被覆盖，从新进程的入口地址开始执行。它只是用另外一个程序替代了当前进程的正文，数据，堆和栈



*Let us
discuss !*



谢谢!