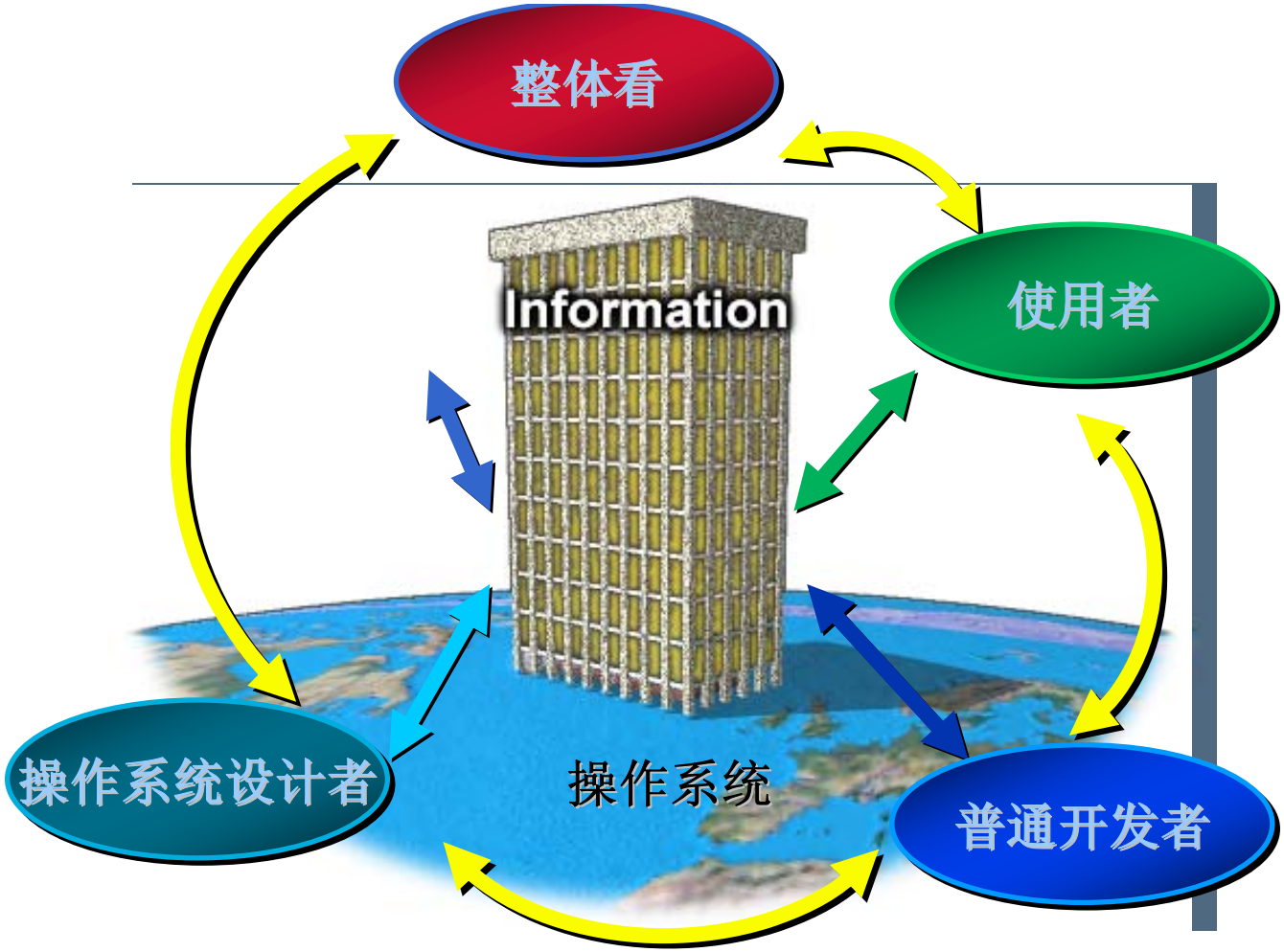
第一章 操作系统概述

- 认识操作系统
- 操作系统的发展
- 开放源代码的Unix/Linux操作系统
- Linux内核
- Linux内核源代码

认识操作系统



不同角度看到的操作系统

认识操作系统－从使用者的角度看

- 打开计算机，首先跳入眼帘的是什么？
- 要拷贝一个文件，具体的拷贝操作是谁完成的？
  - 你需要知道文件存放在何处吗？
  - 柱面、磁道、扇区描述什么？
  - 数据的搬动过程怎样进行
- 繁琐留给自己，简单留给用户
  - 操作系统穿上华丽的外衣－图形界面
  - 操作系统穿上朴素的外衣－字符界面

认识操作系统—从程序开发者的角度看

- 拷贝命令的C语言实现片断

```
inf=open(“/floppy/TEST”,O_RDONLY,0);
out=open(“/mydir/test”,O_WRONLY,0600);
  do{
l=read(inf,buf,4096);
write(outf,buf,l);
} while(l);
close(outf);
close(inf);
```

认识操作系统—从所处位置看

浏览器 信息管理 文件管理系统 游戏

编译程序 编辑程序 命令 解释程序

操作系统

内核

CPU、内存、I/O接口

硬件

认识操作系统－从程序执行看

★操作系统是其它所有用户程序运行的基础。

```
#include<stdio.h>
main()
{
printf(“ Hello
world\n”)
}
```

该程序的执行过程简述如下：

- 操作系统检查字符串的位置是否正确
- 操作系统找到字符串被送往的设备
- 操作系统将字符串送往输出设备窗口系统确定这是一个合法的操作，然后将字符串转换成像素
- 窗口系统将像素写入存储映像区
- 视频硬件将像素表示转换成一组模拟信号控制显示器（重画屏幕）
- 显示器发射电子束。你在屏幕上看到Hello world。

从中看到什么

< >

认识操作系统—从设计者角度看

★从操作系统设计者的角度看

- 操作系统的设计目标是什么？
  - 尽可能地方便用户使用计算机
  - 让各种软件资源和硬件资源高效而协调地运转起来。
- 计算机的硬件资源和软件资源各指什么？
- 假设在一台计算机上有三道程序同时运行，并试图在一台打印机上输出运算结果，必须考虑哪些问题？
- 从操作系统设计者的角度考虑，一个操作系统必须包含以下几部分
  - 操作系统接口
  - CPU管理
  - 内存管理
  - 设备管理
  - 文件管理

认识操作系统—定义

操作系统是计算机系统中的一个系统软件，是一些程序模块的集合——它们能以尽量有效、合理的方式组织和管理计算机的软硬件资源，合理的组织计算机的工作流程，控制程序的执行并向用户提供各种服务功能，使得用户能够灵活、方便、有效的使用计算机，使整个计算机系统能高效、顺畅地运行。

操作系统的发展

★操作系统的演变

❖单道批处理系统

- 串行执行预先组织好的一组任务
- 提高了系统效率 。

❖多道批处理系统

- 可以交错运行多个程序
- 再次提高系统效率。

❖分时系统

- 将处理器的运行时间分成数片，均分或依照一定权重派发给系统中的用户使用
- 快速响应

硬件角度下的操作系统发展轨迹

年 代	硬 件 特点	操作系统特点	背 景
第二代计算机 50年代末~60年代中期 晶体管计算机	1) 采用印刷电路 2) 稳定性与可靠性大大提高 3) 批量生产成为可能 4) 进入实际应用领域但数量有限	1) 单道批处理系统 2) 操作系统以监督软件形式出现 3) 任务按顺序方式处理	1947年发明晶体管
第三代计算机 60年代中期~70年代初 集成电路计算机	1) 体积减小，性价比迅速提高 2) 小型计算机发展迅速 3) 进入商业应用 4) 尚不适合家庭应用的需求	1) 涌现大批操作系统 多道批处理系统、分时系统和实时系统 2) 奠定了现代操作系统的基本框架	1958年发明集成电路 1971年INTEL发明微处理器

硬件角度下操作系统发展的分析

- 在硬件的性价比较低的时候，操作系统设计追求什么？
- 在硬件性价比越来越高后，操作系统的设计开始追求的目标是什么？
- 计算机开始普及后，操作系统的设计开始追求？
- 从第三代到第四代计算机，操作系统的发展逐渐摆脱追随硬件发展的状况，形成自己的理论体系
- 进入第四代系统后，分布式系统和多处理器系统虽然极大的扩充了操作系统理论，但系统结构并没有变化，只是各功能模块得以进一步完善。

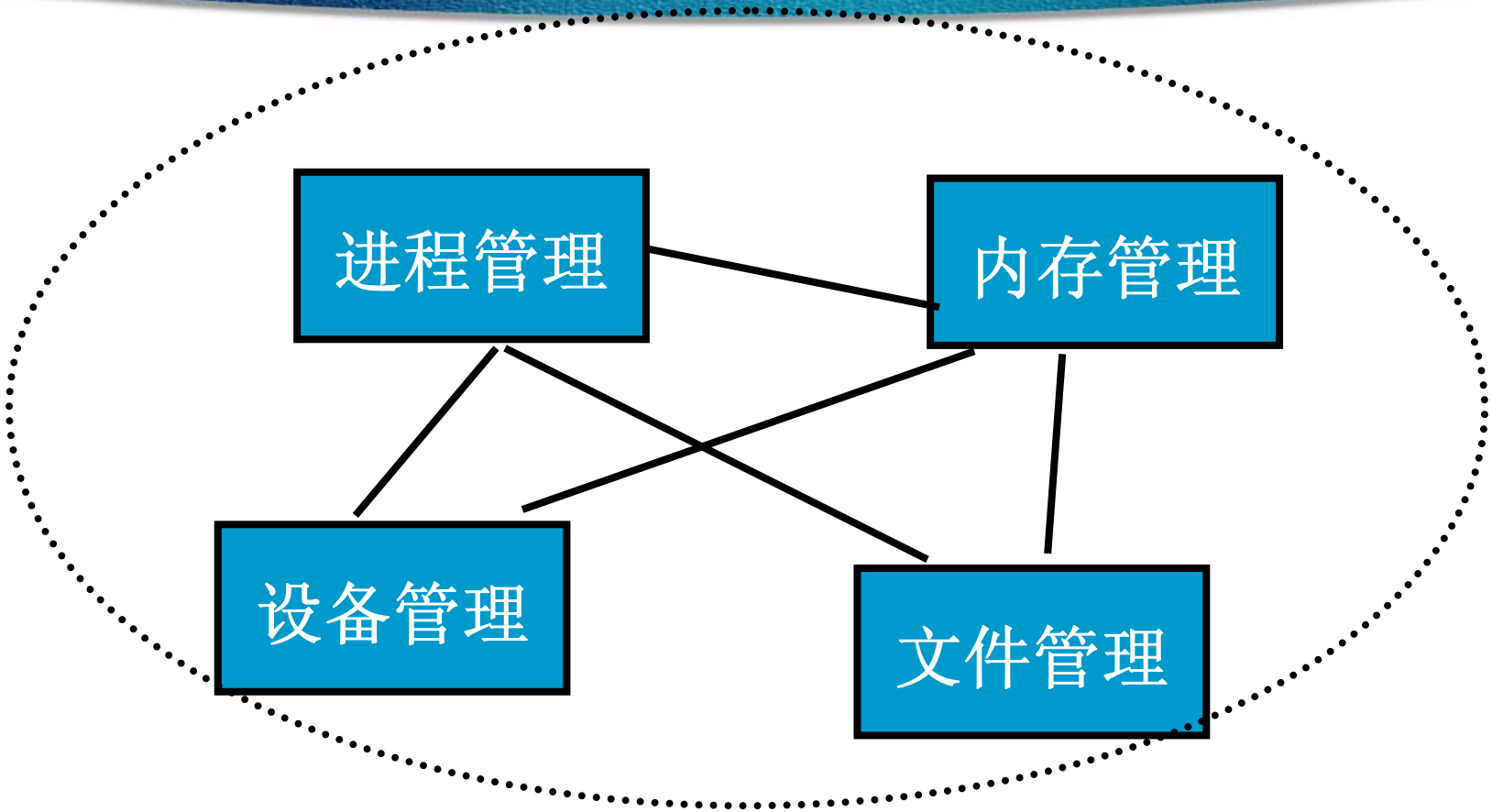

软件角度下的操作系统发展轨迹

主流操作系统	系统特点	计 算 机 语 言	背 景
类Unix系列 WINDOWS系列	人机交互成为主题 1）可视化界面 2）多媒体技	面向对象语言成为主流	80年代中期开始面向对象技术逐步发展
网络操作系统 分布式操作系统	微内核技术兴起	1）JAVA语言 2）脚本语言兴起	1995年JAVA推出
嵌入式系统	单内核与微内核竞争激烈	编程工具向跨平台方向发	1991年免费的操作系统Linux发布

软件角度下的操作系统发展轨迹分析

- 程序设计理论约束着操作系统设计。操作系统的发展滞后于计算机语言的发展，从结构化设计到对象化设计，操作系统总是最后应用新编程理论的软件之一。
- 至今操作系统对于是否需要彻底对象化（即微内核化），还处于徘徊时期，仍在探索单内核与微内核的最佳结合方式。
- 人机交互技术主要是为用户考虑，这是对操作系统设计进行的变革。
- 以Linux为代表的开源软件的出现，打破了带有神秘色彩的传统的封闭式开发模式。

讲究效率的单模块操作系统

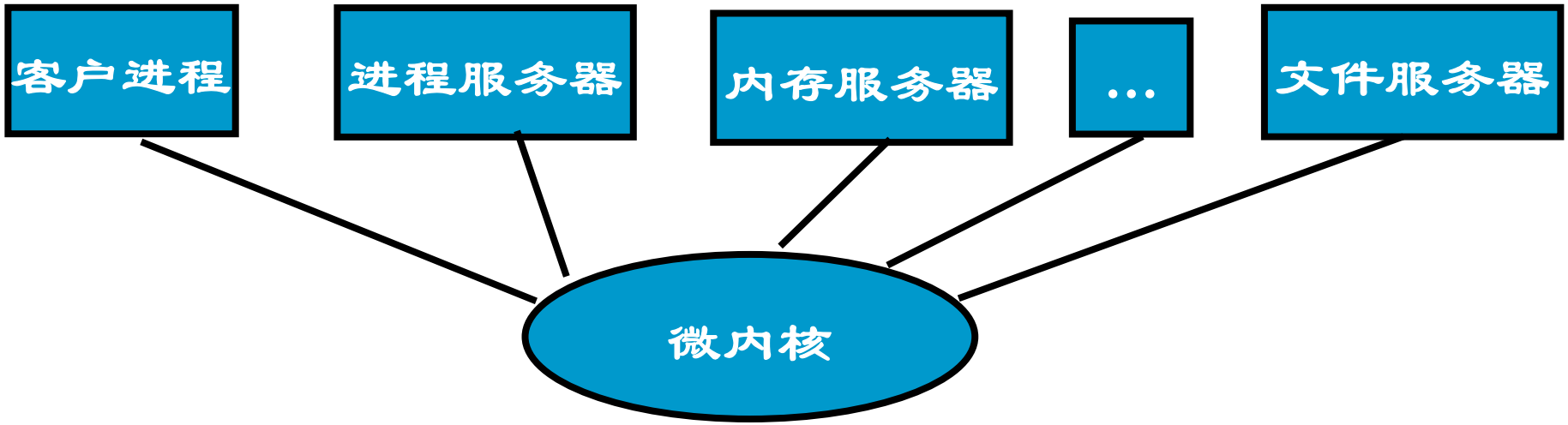


模块之间可以互相调用的单模块结构

讲究效率的单模块操作系统

- 模块之间直接调用函数，除了函数调用的开销外，没有额外开销。
- 庞大的操作系统有数以千计的函数
- 复杂的调用关系势必导致操作系统维护的困难

追求简洁的微内核操作系统



追求简洁的微内核操作系统

- 内核与各个服务器之间通过通信机制进行交互，这使得微内核结构的效率大大折扣。

- 内核发出请求，服务器做出应答

- 为各个服务器模块的相对独立性，使得其维护相对容易

❖在MULTICS（1969） 的肩上

★制研制者Ken Thompson和Dennis M. Ritchie

★站Unix的诞生还伴有C语言呱呱落地

Unix是现代操作系统的代表：安全、可靠、强大的计算能力

Unix的商业化是一把双刃剑

自由而奔放的黑马－Linux

- 诞生于学生之手
- 成长于Internet
- 壮大于自由而开放的文化

Linux之父-Linus Torvalds

- 芬兰、赫尔辛基大学、1990
- 起始于写两个进程
- 然后写驱动程序、文件系统、任务切换程序，从而形成一个操作系统邹形

Linux得以流行的原因之一 —遵循POSIX标准

- POSIX 表示可移植操作系统接口 (Portable Operating System Interface)
- POSIX是在Unix标准化过程中出现的产物。
- POSIX 1003.1标准定义了一个最小的Unix操作系统接口
- 任何操作系统只有符合这一标准，才有可能运行Unix程序

Linux的肥沃土壤－GNU

- GNU 是 GNU Is Not Unix 的递归缩写，是自由软件基金会的一个项目 。
- GNU 项目产品包括 emacs 编辑器、著名的 GNU C 和 Gcc编译器等，这些软件叫做GNU软件。
- GNU 软件和派生工作均适用 GNU 通用公共许可证，即 GPL（General Public License ）
- Linux的开发使用了众多的GUN工具

GPL－开源软件的法律

- GPL 允许软件作者拥有软件版权
- 但GPL规定授予其他任何人以合法复制、发行和修改软件的权利。

Linux系统或发布版

- 符合 POSIX 标准的操作系统内核、 Shell 和外围工具。
- C 语言编译器和其他开发工具及函数库
- X Window 窗口系统
- 各种应用软件，包括字处理软件、图象处理软件等。

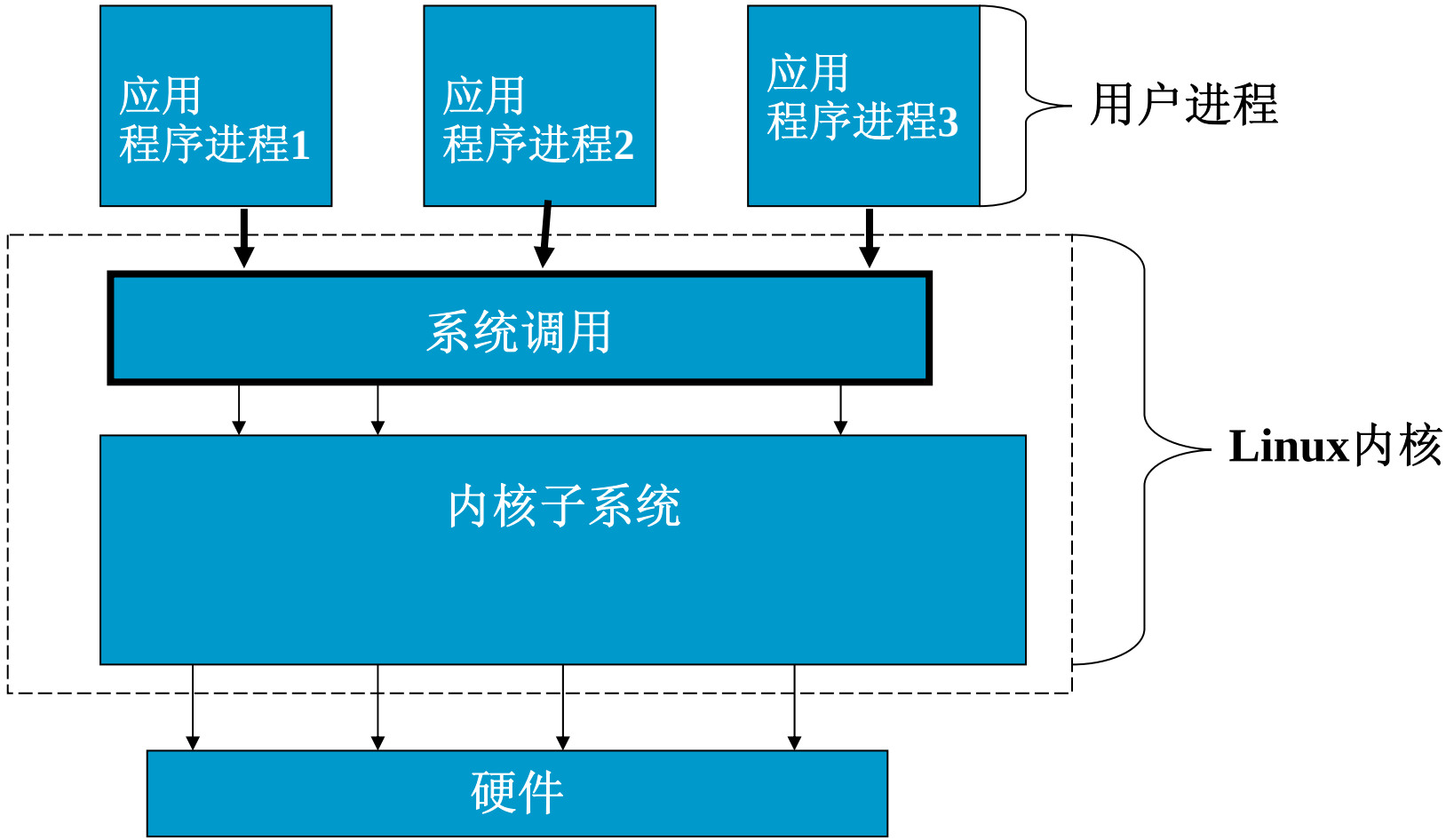
开放与协作的开发模式

- 世界各地软件爱好者集体智慧的结晶
- 提供源代码，遵守GPL。
- 经历了各种各样的测试与考验，软件的稳定性好。
- 开发人员凭兴趣去开发，热情高，具有创造性。

Linux内核

- Linus领导下的开发小组开发出的系统内核
- 是所有Linux 发布版本的核心
- 内核开发人员一般在百人以上，任何自由程序员都可以提交自己的修改工作。
- 采用邮件列表来进行项目管理、交流、错误报告
- 有大量的用户进行测试，正式发布的代码质量高

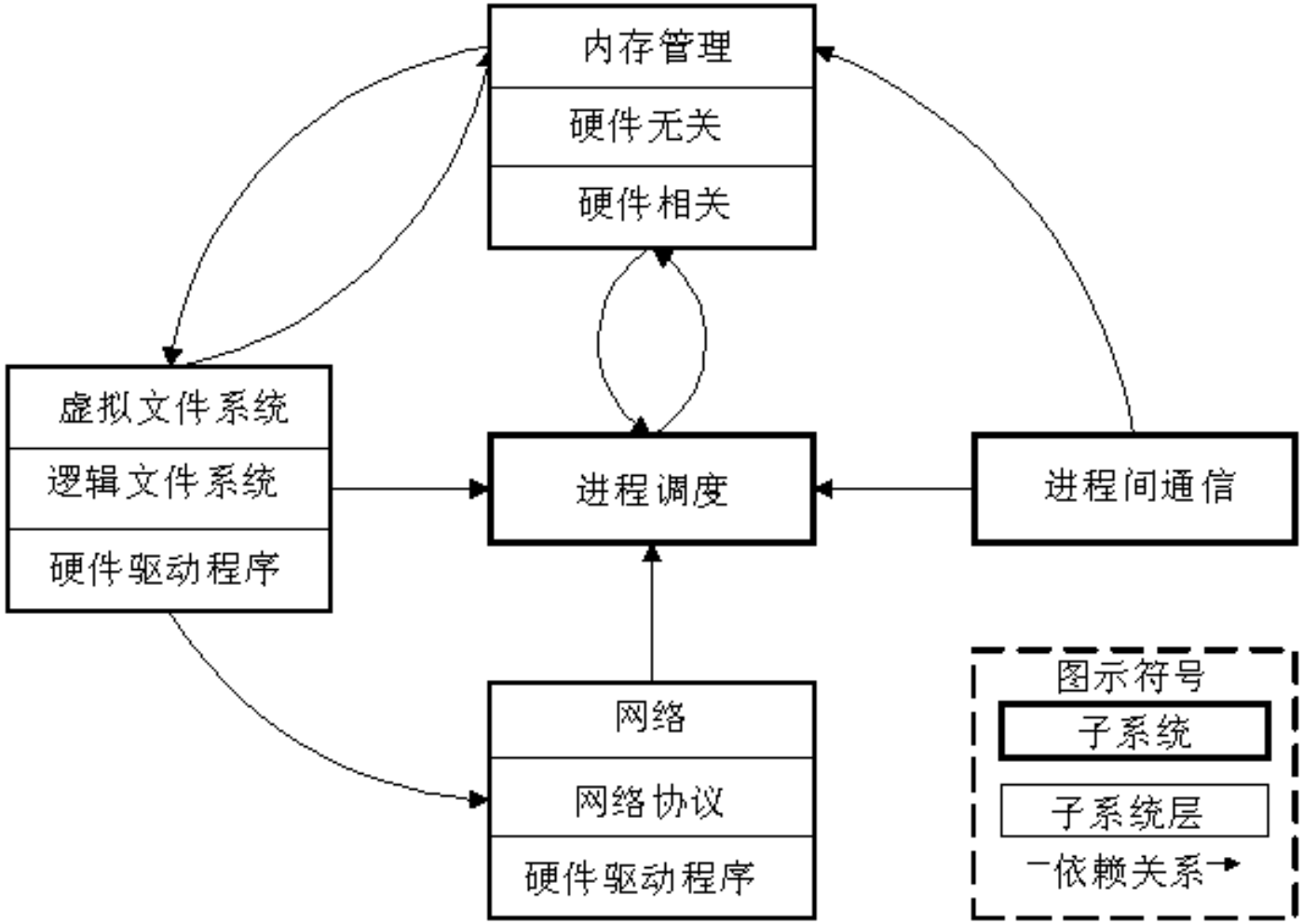

整个系统的核心—内核



整个系统的核心—内核

- 用户进程—运行在Linux内核之上的一个庞大软件集合。
- 系统调用—内核的出口，用户程序通过它使用内核提供的功能。
- Linux内核—操作系统的灵魂，负责管理磁盘上的文件、内存，负责启动并运行程序，负责从网络上接收和发送数据包等等。
- 硬件—包括了Linux安装时需要的所有可能的物理设备。例如，CPU、 内存、硬盘、网络硬件等等。

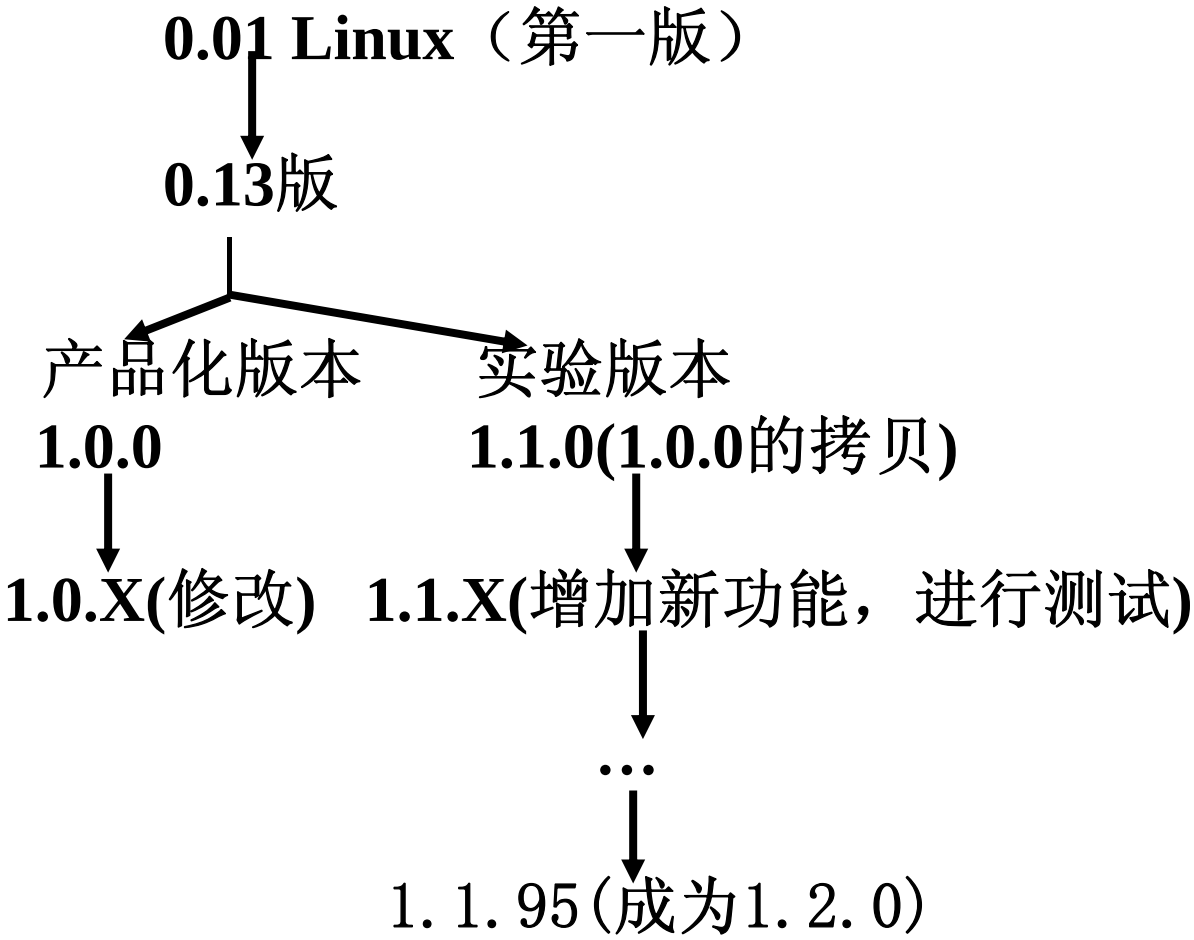
内核子系统


内存管理
硬件无关
硬件相关
虚拟文件系统
逻辑文件系统
硬件驱动程序
进程调度
进程间通信
网络
网络协议
硬件驱动程序
图示符号
子系统
子系统层
依赖关系

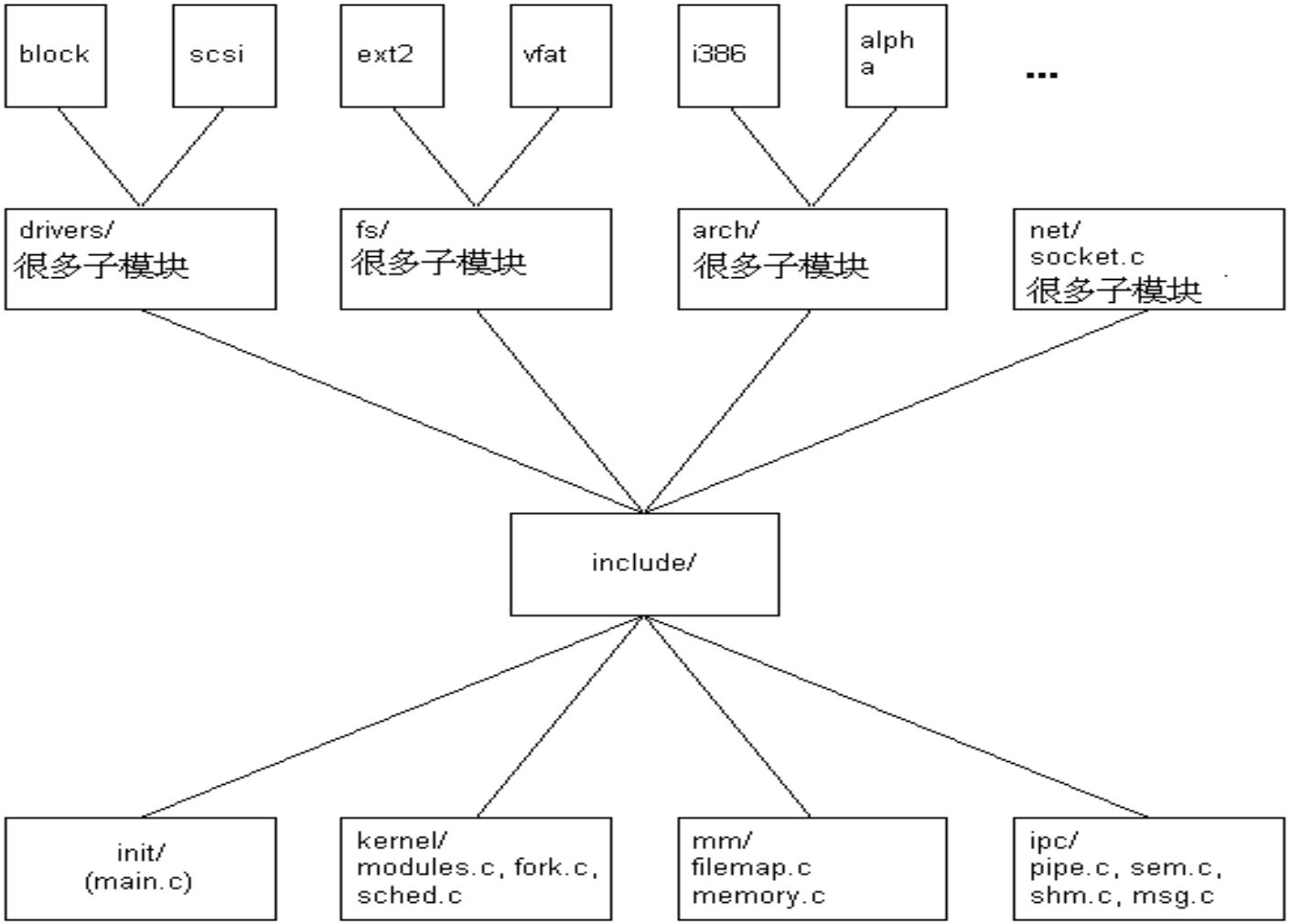
内核子系统

- 进程调度－控制着进程对CPU的访问。
- 内存管理－允许多个进程安全地共享主内存区域
- 虚拟文件系统－隐藏各种不同硬件的具体细节，为所有设备提供统一的接口。
- 网络－提供了对各种网络标准协议的存取和各种网络硬件的支持。
- 进程间通信(IPC)－ 支持进程间各种通信机制，包括共享内存、消息队列及管道等。

Linux内核版本树


0.01 Linux（第一版）
0.13版
产品化版本
实验版本
1.0.0
1.1.0(1.0.0的拷贝)
1.0.X(修改)
1.1.X(增加新功能，进行测试)
…
1.1.95(成为1.2.0)

内核源代码结构


block
scsi
ext2
vfat
i386
alph
a
...
drivers/
很多子模块
fs/
很多子模块
arch/
很多子模块
net/
socket.c
很多子模块
include/
init/
(main.c)
kernel/
modules.c, fork.c,
sched.c
mm/
filemap.c
memory.c
ipc/
pipe.c, sem.c,
shm.c, msg.c

Linux内核源代码分析工具

- Linux超文本交叉代码检索工具
  http://lxr.linux.no/

- **Windows平台下的源代码阅读工具Source Insight**

“内核之旅”网站

- http://www.kerneltravel.net/
- 电子杂志栏目是关于内核研究和学习的资料
- 第一期“走入Linux世界”涉猎了操作系统的来龙去脉后与大家携手步入Linux世界。
- 下载代码，亲手搭建实验系统。

第二章 内存寻址

- 内存寻址的演变
- 段机制
- 分页机制
- Linux中的汇编语言
- Linux系统地址映射示例

内存寻址—操作系统设计的硬件基础之一

- 操作系统—横跨软件和硬件的桥梁
- 内存寻址—操作系统设计的硬件基础之一
- 操作系统的设计者必须在硬件相关的代码与硬件无关的代码之间划出清楚的界限，以便于一个操作系统很容易地移植到不同的平台。
- 在这众多的平台中，大家最熟悉的就是i386，即Intel80386体系结构。因此，我们所介绍的内存寻址也是以此为背景。

内存寻址的不同时期

- 石器时期－8位
- 青铜时期－16位
- 白银时期－24位
- 黄金时期－32位

石器时期—8位寻址

- 在微处理器的历史上，第一款微处理器芯片4004是由Intel推出的，4位。
- 在4004之后，intel推出了一款8位处理器叫8080，它有1个主累加器（寄存器A）和6个次累加器（寄存器B,C,D,E,H和L）
- 那时没有段的概念，访问内存都要通过绝对地址，因此程序中的地址必须进行硬编码（给出具体地址），而且也难以重定位

青铜时期—“段”的引入

- intel开发出的16位的处理器叫8086，标志着Intel X86王朝的开始，同时引入了“段 ”概念。
- 段描述了一块有限的内存区域，区域的起始位置存在专门的寄存器（段寄存器）中。
- 8086处理器地址线扩展到了20位，寻址空间到了1M
- 也就是把1M大的空间分成数个64k的段来管理（化整为零了）。
- 把16位的段地址左移动4位后，再与16位的偏移量相加便可获得一个20位的内存地址，

白银时期—“保护模式”的引入

- intel的80286处理器于1982年问世 。
- 地址总线位数增加到了24位 。
- 从此开始引进了一个全新理念—保护模式
- 访问内存时不能直接从段寄存器中获得段的起始地址了，而需要经过额外转换和检查 。
- 80286处理器一些致命的缺陷注定不能长久，它很快被天资卓越的兄弟——80386代替了

黄金时期—内存寻址的飞跃

- 80386是一个32位的CPU，其寻址能力达到4GB
- Intel选择了在段寄存器的基础上构筑保护模式，并且保留段寄存器16位
- 在保护模式下，它的段范围不再受限于64K，可以达到4G
- 这真正解放了软件工程师，他们不必再费尽心思去压缩程序规模，软件功能也因此迅速提升
- 从80386以后，Intel的CPU经历了80486、Pentium、PentiumII、PentiumIII等型号，但基本上属于同一种系统结构的改进与加强，而无本质的变化，所以我们把80386以后的处理器统称为IA32（32 Bit Intel Architecture）。

IA32寄存器简介

- 把16位的通用寄存器、标志寄存器以及指令指针寄存器扩充为32位的寄存器
- 段寄存器仍然为16位。
- 增加4个32位的控制寄存器
- 增加4个系统地址寄存器
- 增加8个调式寄存器
- 增加2个测试寄存器

常用寄存器简介

★通用寄存器

- 8个通用寄存器是8086寄存器的超集，它们分别为：EAX，EBX，ECX，EDX，EBP，EBP，ESI及 EDI

★段寄存器

- 8086中有4个16位的段寄存器：CS、DS、SS、ES，分别用于存放可执行代码的代码段、数据段、堆栈段和其他段的基地址。
- 这些段寄存器中存放的不再是某个段的基地址，而是某个段的**选择符**（Selector）
- 段基地址存放在段描述符表（Descriptor ）中，表的索引就是**选择符**

常用寄存器简介

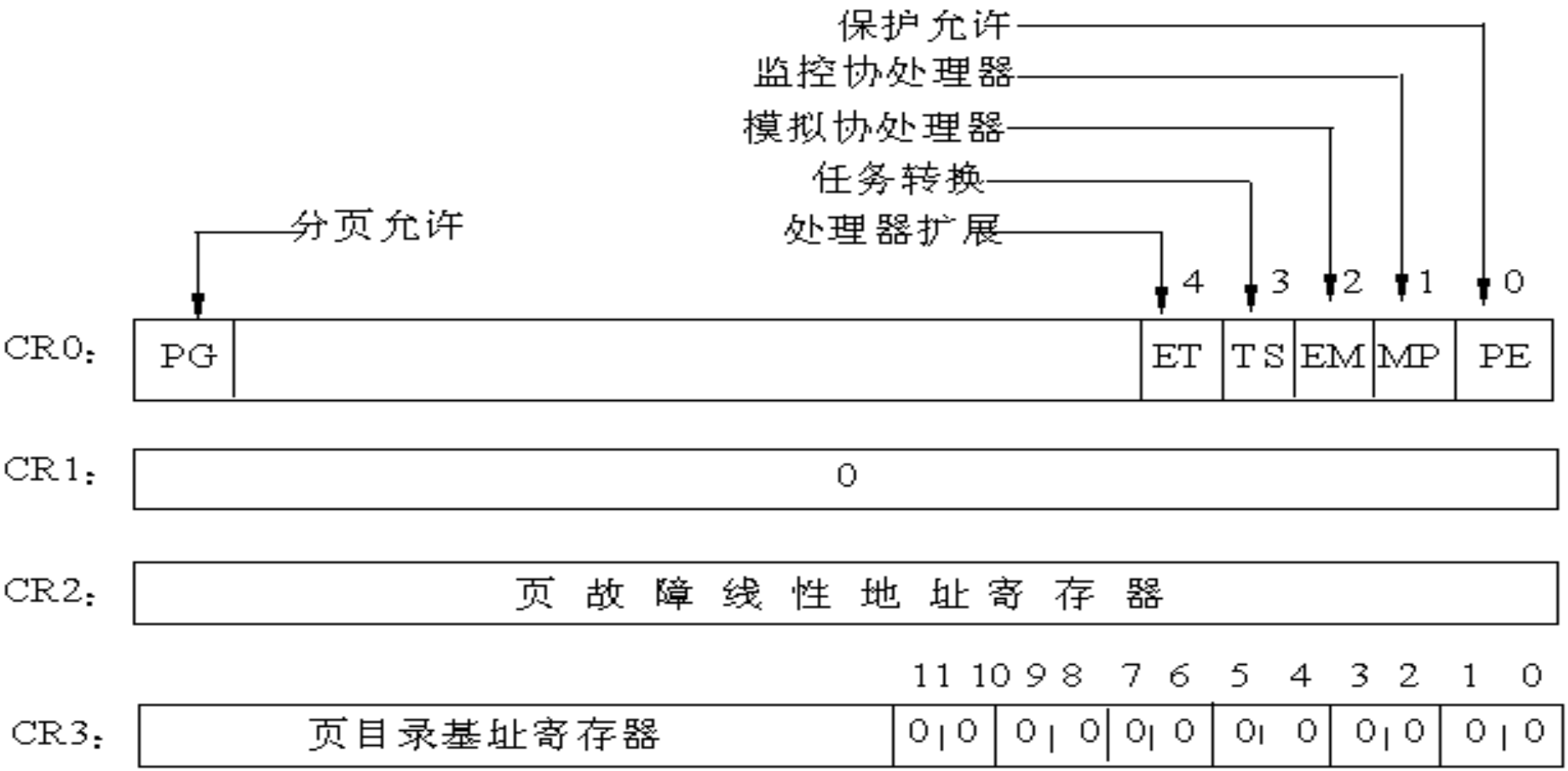
★指令指针寄存器

- ❖指令指针寄存器EIP中存放下一条将要执行指令的偏移量（offset ），这个偏移量是相对于目前正在运行的代码段寄存器CS而言的。偏移量加上当前代码段的基地址，就形成了下一条指令的地址。
- ❖EIP中的低16位可以被单独访问，给它起名叫指令指针IP寄存器，用于16位寻址。

★标志寄存器

- ❖标志寄存器EFLAGS存放有关处理器的控制标志，很多标志与16位FLAGS中的标志含义一样。

用于分页机制的控制寄存器



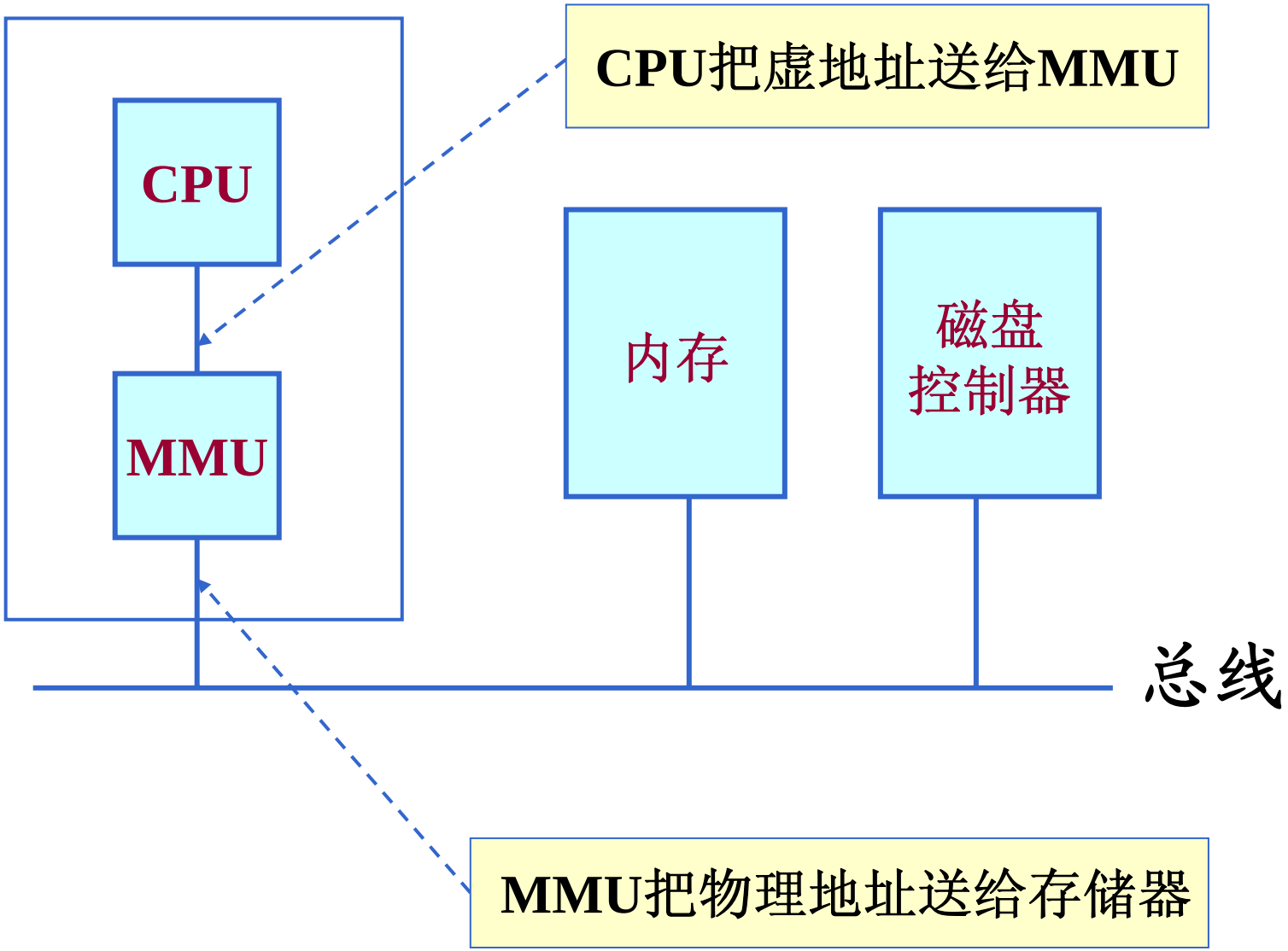
物理地址、虚拟地址及线性地址

★将主板上的物理内存条所提供的内存空间定义为**物理内存空间**，其中每个内存单元的实际地址就是**物理地址**

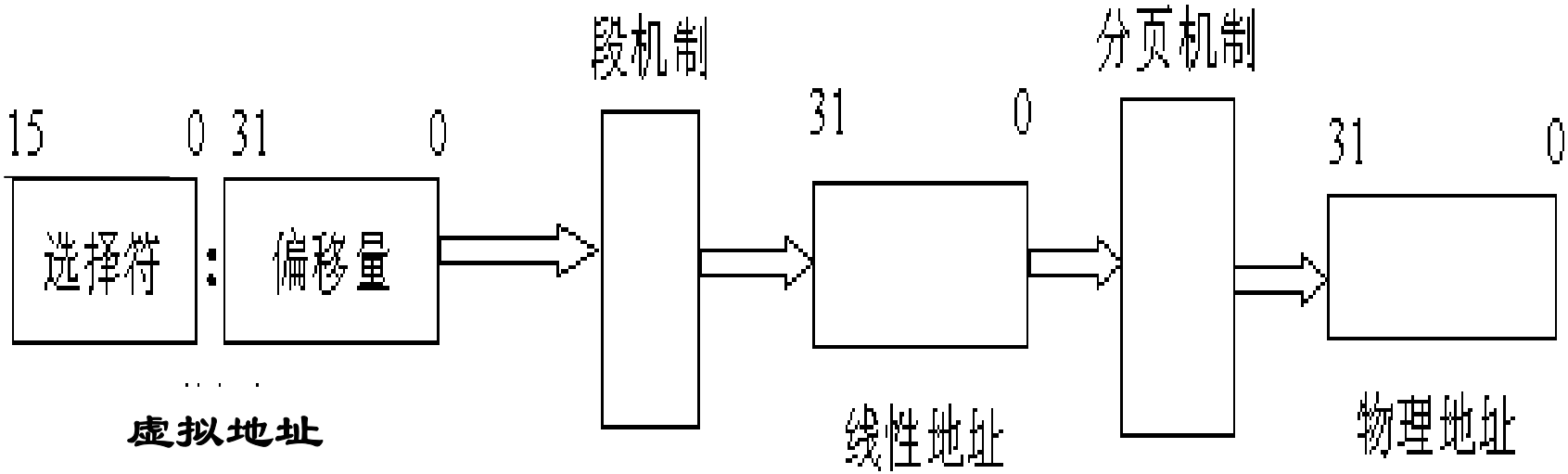
★将应用程序员看到的内存空间定义为虚拟地址空间(或地址空间)，其中的地址就叫**虚拟地址(或虚地址)**，一般用“**段：偏移量**”的形式来描述

★**线性地址空间**是指一段连续的，不分段的，范围为0到4GB的地址空间，一个**线性地址**就是线性地址空间的一个绝对地址。

地址之间的转换—保护模式下的寻址



地址之间的转换—MMU机制



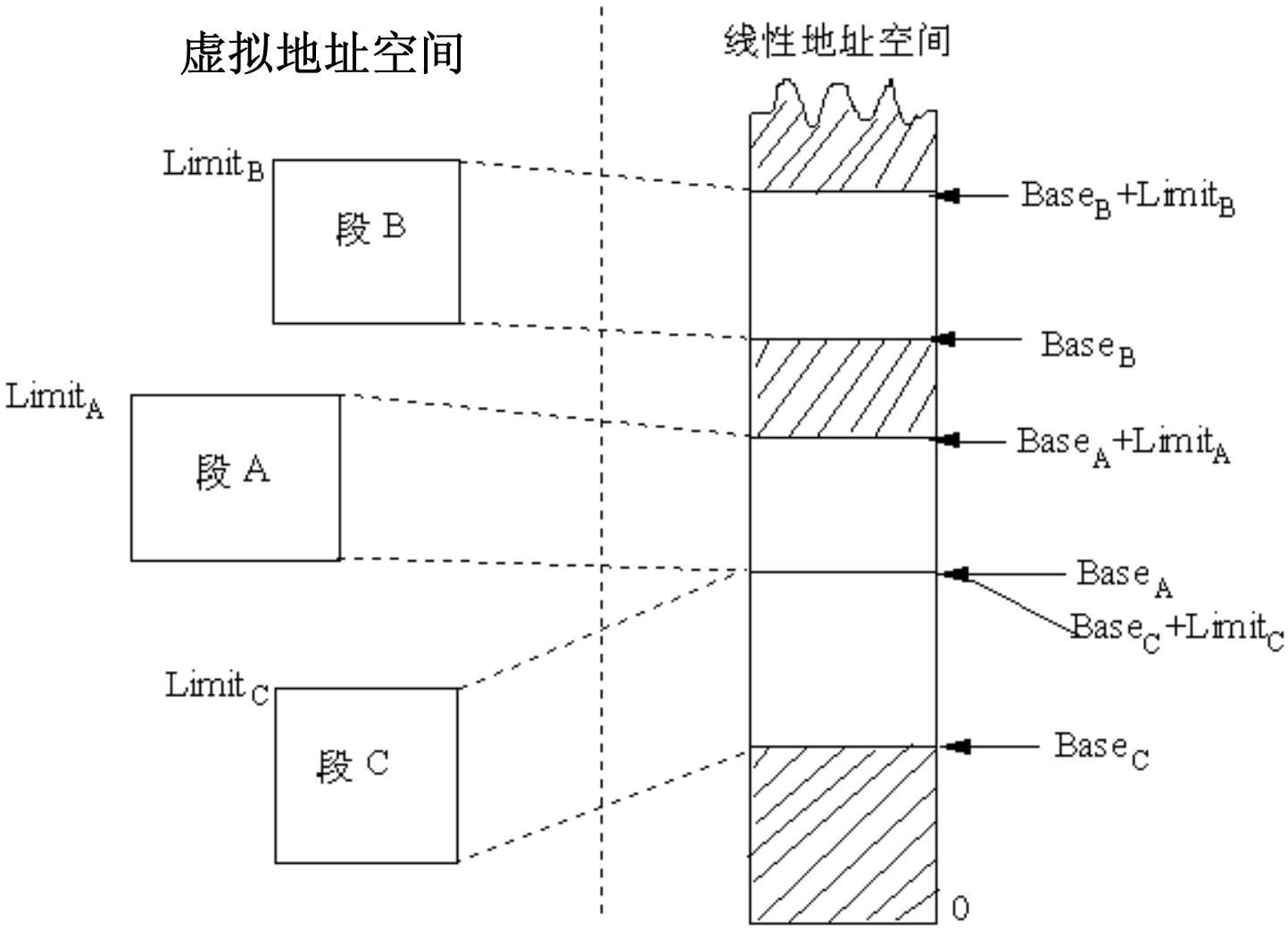
段机制

★段是虚拟地址空间的基本单位，段机制必须把虚拟地址空间的一个地址转换为线性地址空间的一个线性地址。

★用三个方面来描述段

- 段的基地址(Base Address)：在线性地址空间中段的起始地址。
- 段的界限(Limit)：在虚拟地址空间中，段内可以使用的最大偏移量。
- 段的保护属性(Attribute)：表示段的特性。例如，该段是否可被读出或写入，或者该段是否作为一个程序来执行，以及段的特权级等等。

虚拟—线性地址的转换


虚拟地址空间
线性地址空间
Limit B
段 B
Limit A
段 A
Limit C
段 C
Base B +Limit B
Base B
Base A +Limit A
Base A
Base C +Limit C
Base C
0

段描述符表—段表

★如图所示的段描述符表（或叫段表）来描述转换关系。段号描述的是虚拟地址空间段的编号，基地址是线性地址空间段的起始地址。

★段描述符表中的每一个表项叫做段描述符

索引（段号）	基地址	界限	属性
0	$Base_b$	$Limit_b$	$Attribute_b$
1	$Base_a$	$Limit_a$	$Attribute_a$
2	$Base_c$	$Limit_c$	$Attribute_c$

保护模式下的其他描述符表简介

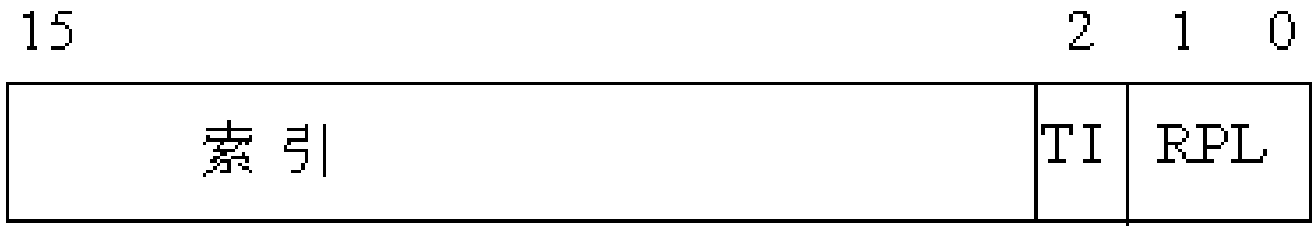
- ★全局描述符表GDT (Gloabal Descriptor Table)
- ★中断描述符表IDT (Interrupt Descriptor Table)
- ★局部描述符表LDT (Local Descriptor Table)
- ★为了加快对这些表的访问，Intel设计了专门的寄存器，以存放这些表的基地址及表的长度界限 。这些寄存器只供操作系统使用。

有关这些表的详细内容请参看有关保护模式的参考书。

保护模式下段寄存器中存放什么

★存放索引或叫段号，因此，这里的段寄存器也叫选择符，即从描述符表中选择某个段。

★选择符（段寄存器）的结构：



★RPL表示请求者的特权级（Requestor Privilege Level）

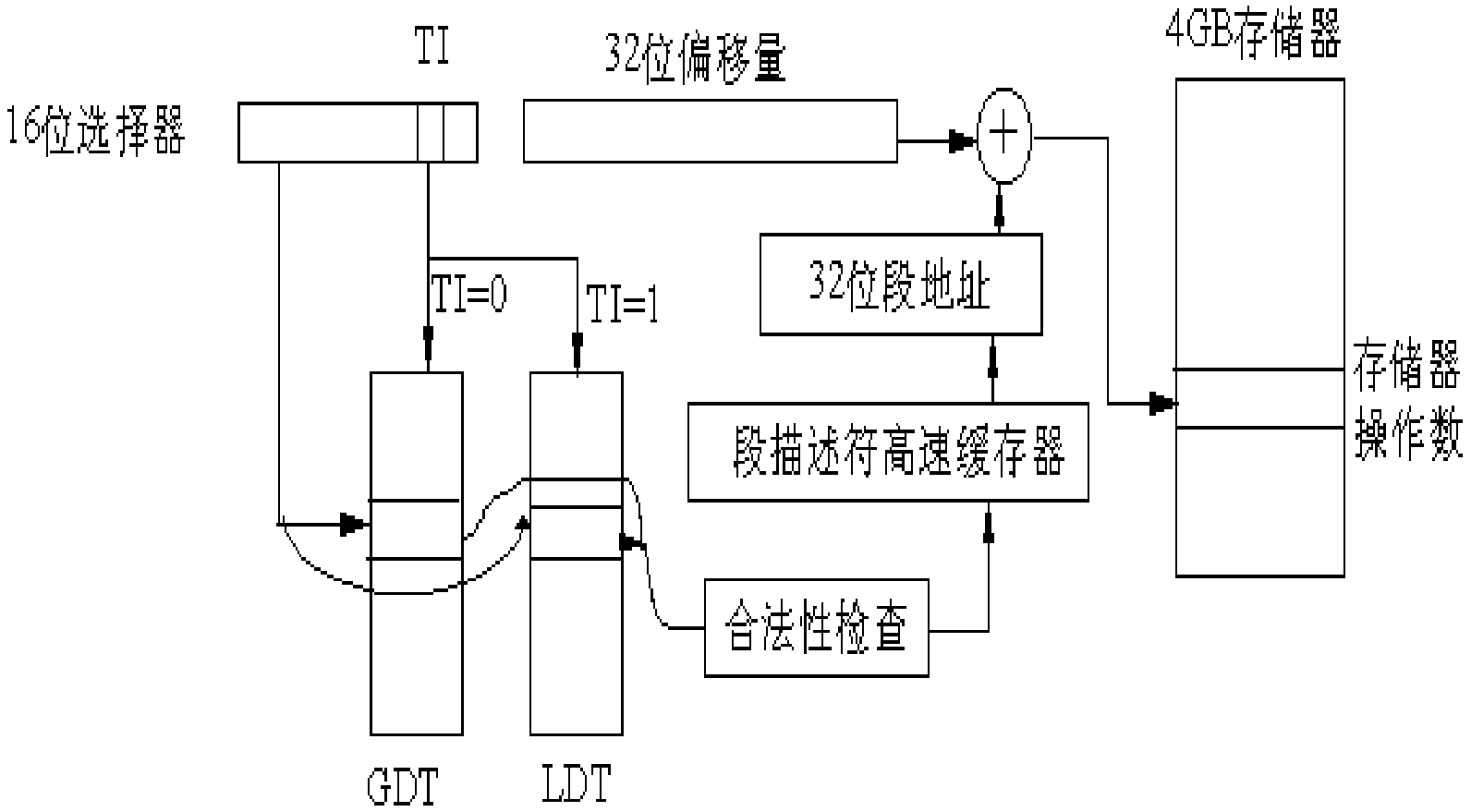
保护模式下的特权级

★保护模式提供了四个特权级，用0~3四个数字表示

★很多操作系统（包括Linux,Windwos）只使用了其中的最低和最高两个，即0表示最高特权级，对应内核态；3表示最低特权级，对应用户态。

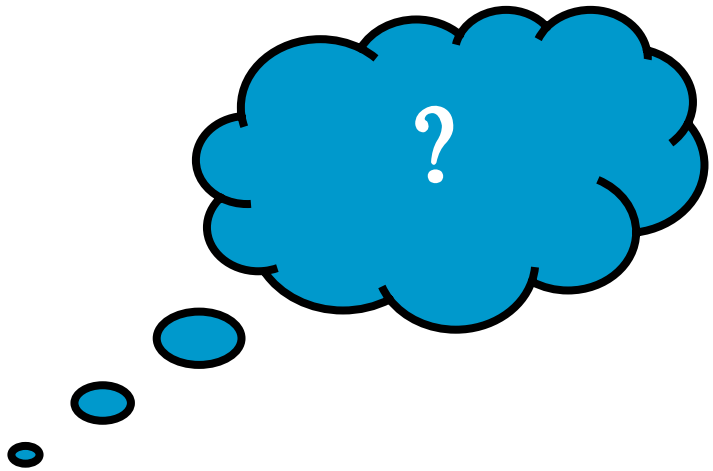
★保护模式规定，高特权级可以访问低特权级，而低特权级不能随便访问高特权级。

地址转换及保护


TI
32位偏移量
4GB存储器
16位选择器
TI=0
TI=1
32位段地址
存储器
操作数
段描述符高速缓存器
合法性检查
GDT
LDT

Linux中的段

★Linux是怎样处理段机制？

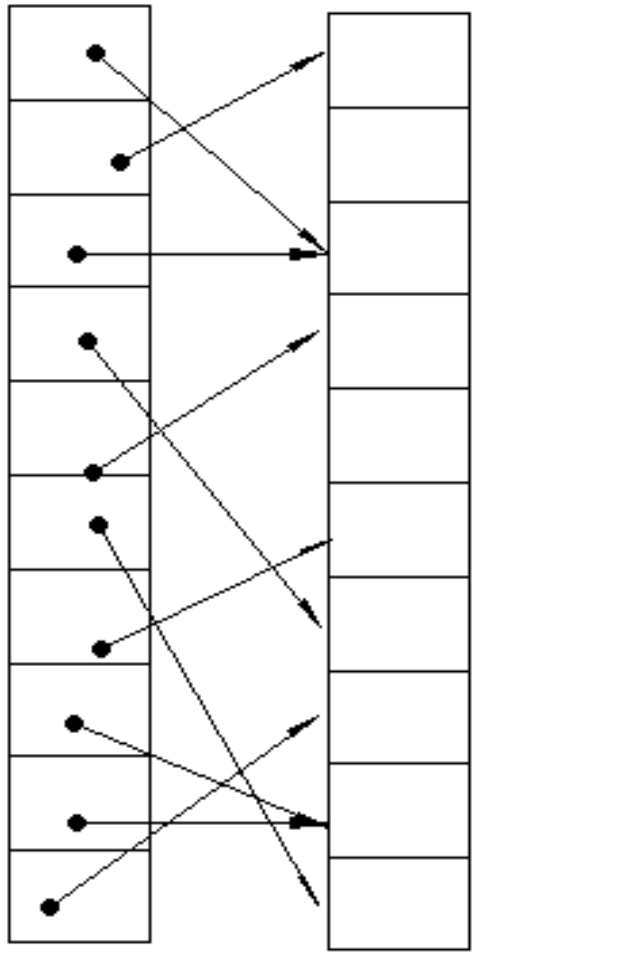


分页机制—页

★将线性地址空间划分成若干大小相等的片，称为**页**（Page）

★物理地址空间分成与**页大小相等**的若干存储块，称为（**物理**）**块或页面**（Page Frame）

★页的大小应该为多少？由谁确定？



线性地址空间　　物理地址空间

分页机制—页表

★页表是把线性地址映射到物理地址的一种数据结构。

★页表中应当包含如下内容：

- ❖物理页面基地址：线性地址空间中的一个页装入内存后所对应的物理页面的起始地址。
- ❖页的属性：表示页的特性。例如该页是否在内存，是否可被读出或写入等。

★页面的大小为4KB，物理页面基地址需要多少位就可以？

分页机制—页表项结构

31	11	0
物理页面基地址	属性	

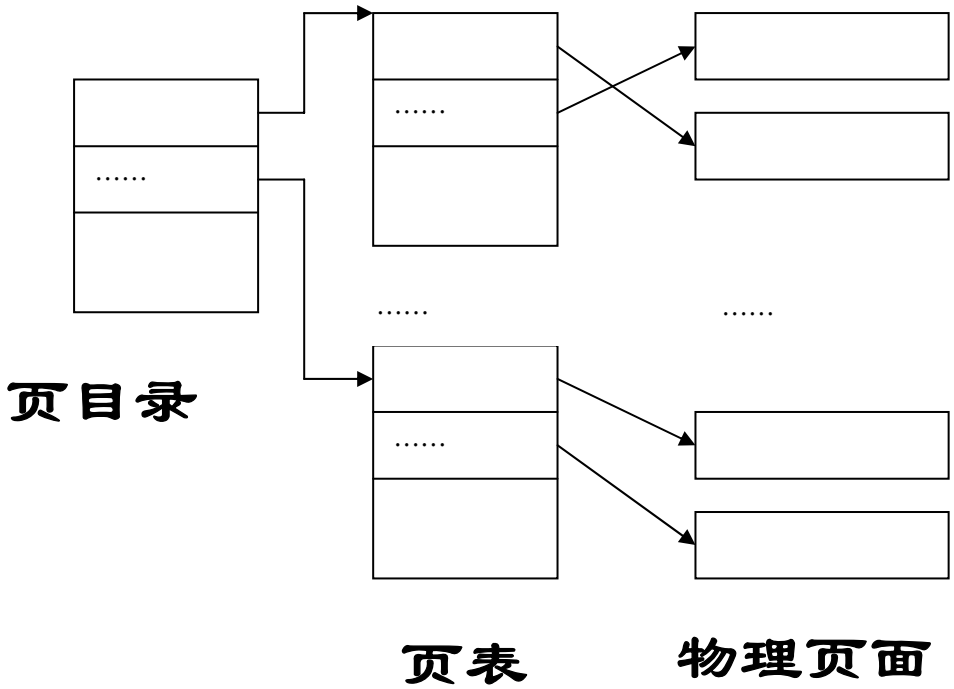
★物理页面基地址： 指的是页所对应的物理页面在内存的起始物理地址。相当于物理块号(为什么?)

★其最低12位全部为0，因此用高20位来描述32位的地址。

★属性见书

分页机制—两级页表

★为什么要采用两级页表？



分页机制—线性地址结构

★这个结构的伪代码描述如下

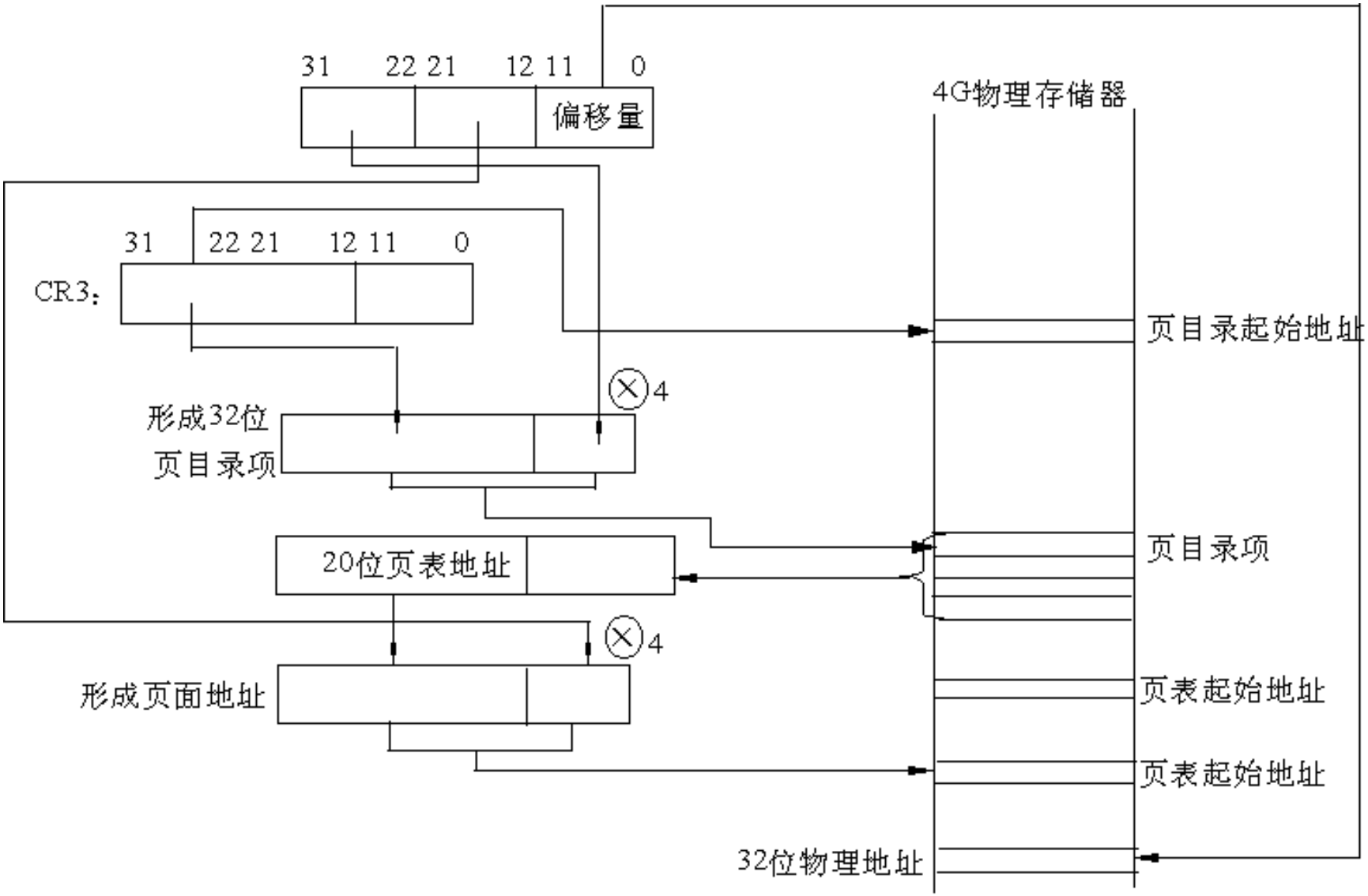
```
typedef struct {
      unsigned int dir:10;      /*用作页目录中的下标，对应的
                                目录项指向一个页表*/
      unsigned int page:10      /*用作页表的下标，对应的页表
                                 项指向一个物理页面*/
      unsigned int offset:12    /*在4K字物理页面内的偏移量*/
} LinearAddr
```

分页机制—硬件保护机制

★对于页表，页的保护是由属性部分的U/S标志和R/W标志来控制的。当U/S标志为0时，只有处于内核态的操作系统才能对此页或页表进行寻址。当这个标志为1时，则不管在内核态还是用户态，总能对此页进行寻址。

★此外，与段的三种存取权限（读、写、执行）不同，页的存取权限只有两种（读、写）。如果页目录项或页表项的读写标志为0，说明相应的页表或页是只读的，否则是可读写的。

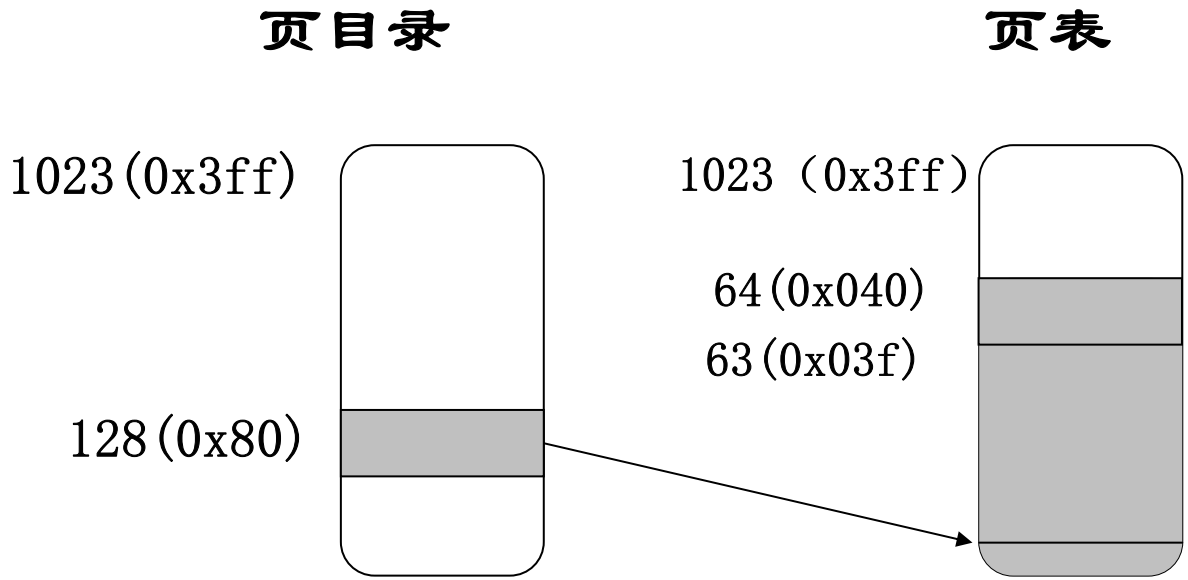
分页机制—线性地址到物理地址的转换



分页机制—分页示例

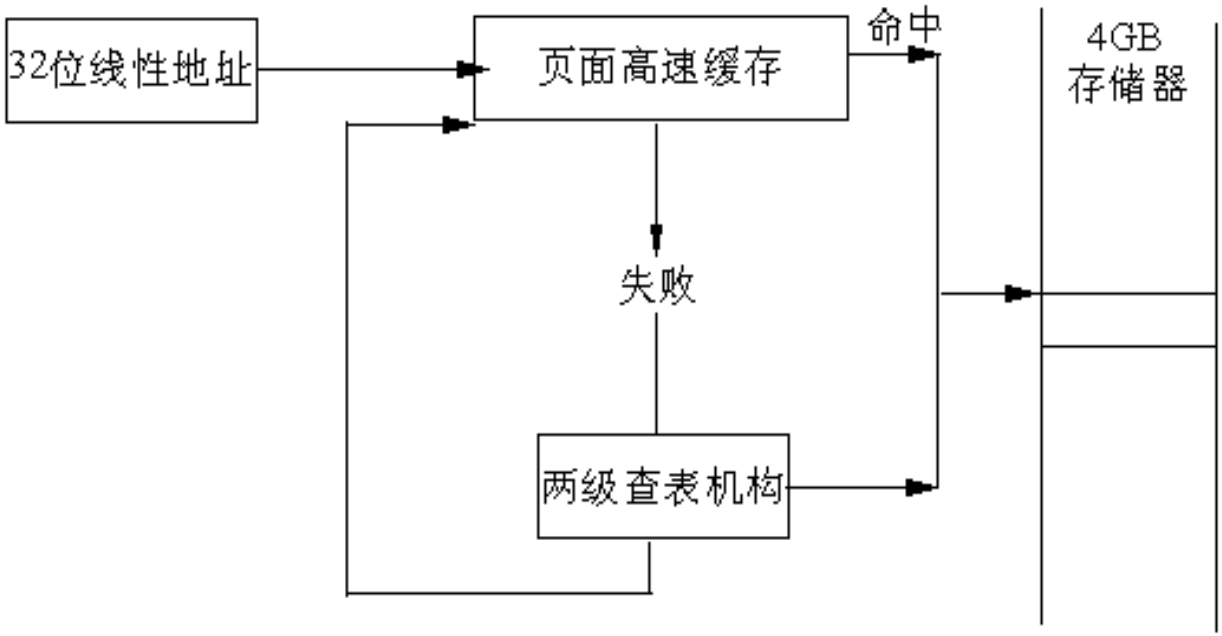
- ★假如操作系统给一个正在运行的进程分配的线性地址空间范围是0x20000000 到 0x2003ffff。这个空间由64页组成。
- ★我们从分配给进程的线性地址的最高10位（分页硬件机制把它自动解释成页目录域）开始。这两个地址都以2开头，后面跟着0，因此高10位有相同的值，即十六进制的0x080或十进制的128。因此，这两个地址的页目录域都指向进程页目录的第129项。相应的目录项中必须包含分配给进程的页表的物理地址，如图2.13。如果给这个进程没有分配其它的线性地址，则页目录的其余1023项都为0，也就是这个进程在页目录中只占一项。

分页机制—分页示例



★假设进程需要读线性地址0x20021406中的内容。这个地址由分页机制如何进行处理?

分页机制—页面高速缓存



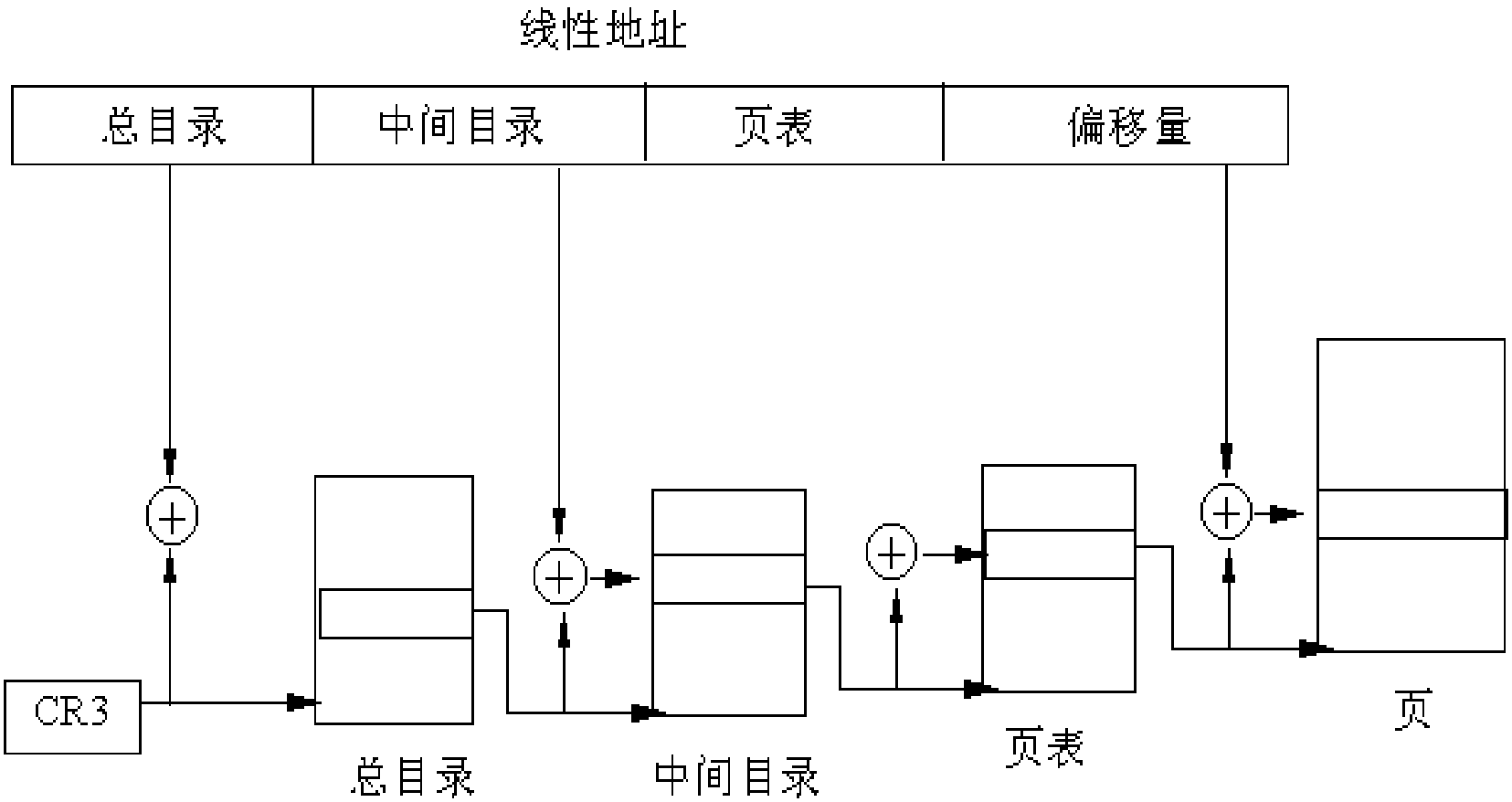
分页机制—Linux中的分页

★Linux主要采用分页机制来实现虚拟存储器管理，因为：

- Linux的分段机制使得所有的进程都使用相同的段寄存器值，这就使得内存管理变得简单，也就是说，所有的进程都使用同样的线性地址空间（0~4G）。
- Linux设计目标之一就是能够把自己移植到绝大多数流行的处理器平台。但是，许多RISC处理器支持的段功能非常有限。

★为了保持可移植性，Linux采用三级分页模式而不是两级

分页机制—Linux中的分页



Linux中的C语言和汇编语言

★ GNU 的C语言

http://www.faqs.org/docs/learnc/

★AT&T的汇编:参见书

Linux系统地址映射示例

★Linux采用分页存储管理。虚拟地址空间划分成固定大小的“页”，由MMU在运行时将虚拟地址映射（变换）成某个物理页面中的地址

★IA32的MMU对程序中的虚拟地址先进行段式映射（虚拟地址转换为线性地址），然后才能进行页式映射（线性地址转换为物理地址）

★Linux巧妙地使段式映射实际上不起什么作用

Linux系统地址映射示例

假定我们有一个简单的C程序Hello.c

```
# include <stdio.h>
greeting ( )
{
    printf("Hello,world!\n");
}
main()
 {
      greeting();
 }
```

Linux系统地址映射示例

用Linux的实用程序objdump对其可执行代码进行反汇编：

```
% objdump -d hello
08048568 <greeting>:
   8048568:     pushl  %ebp
   8048569:     movl  %esp, %ebp
   804856b:     pushl  $0x809404
   8048570:     call    8048474  <_init+0x84>
   8048575:     addl   $0x4, %esp
   8048578:     leave
   8048579:     ret
   804857a:     movl  %esi, %esi
   0804857c <main>:
   804857c:     pushl  %ebp
   804857d:     movl  %esp, %ebp
   804857f:     call    8048568  <greeting>
   8048584:     leave
   8048585:     ret
   8048586:     nop
   8048587:     nop
```

Linux系统地址映射示例

★Linux最常见的可执行文件格式为elf(Executable and Linkable Format)。

★在elf格式的可执行代码中，ld总是从0x8000000开始安排程序的“代码段”，这个地址就是虚地址

★程序执行时在物理内存中的实际地址，则由内核为其建立内存映射时临时分配，具体地址取决于当时所分配的物理内存页面。

本章小节

- 内存寻址的演变
- 段机制
- 分页机制
- Linux中的汇编语言
- Linux系统地址映射示例

“内核之旅”网站

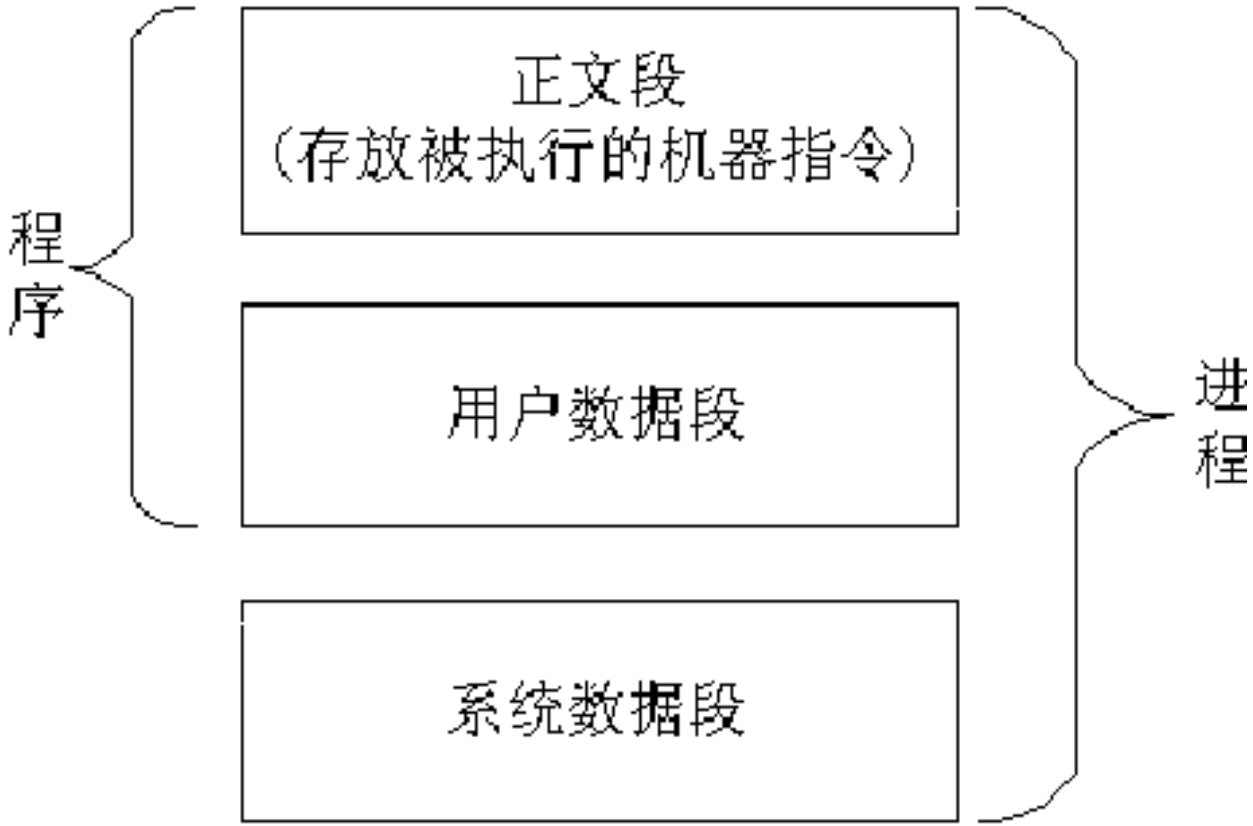
- http://www.kerneltravel.net/
- 电子杂志栏目是关于内核研究和学习的资料
- 第二期“《i386体系结构》分两部分，上半部分让大家认识一下Intel系统中的内存寻址和虚拟内存的来龙去脉。下半部分将实现一个最短小的可启动内核，一是加深对i386体系的了解，再就是演示系统开发的原始过程。

- 下载代码进行调试

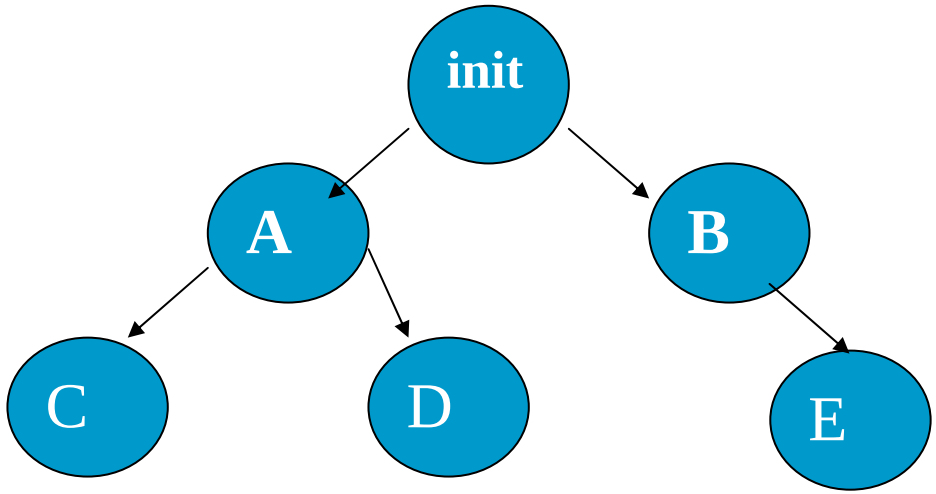
第三章 进程

- 进程介绍
- 进程控制块
- 进程的组织方式
- 进程调度
- 进程的创建
- 与进程相关的系统调用及其应用
- 与调度相关的系统调用及应用

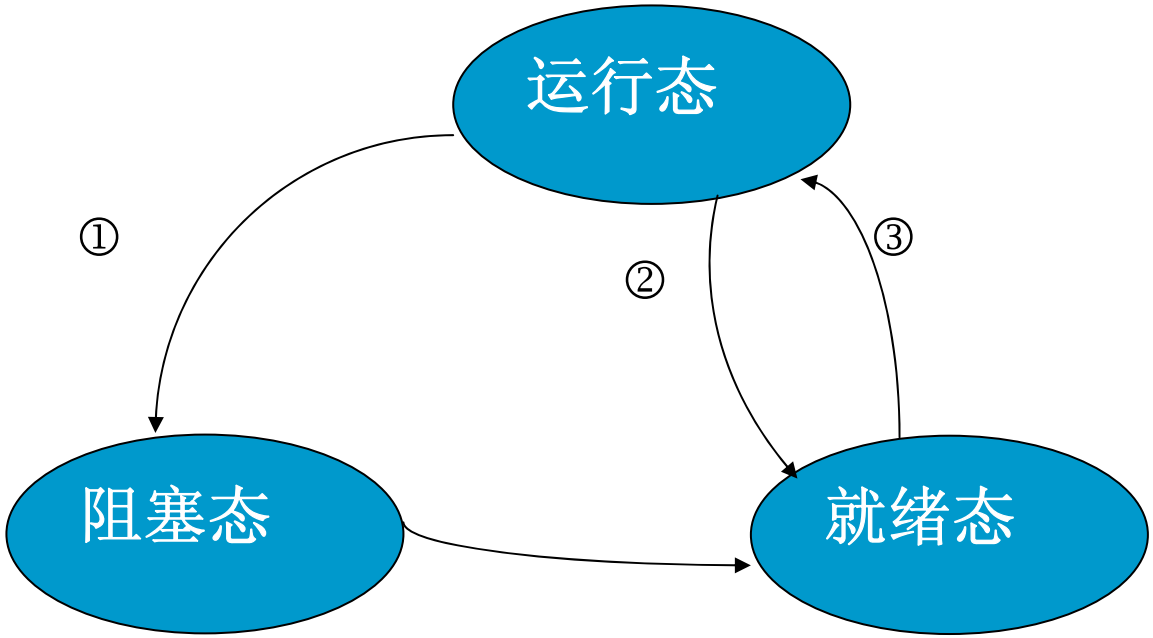
进程介绍—程序和进程



进程介绍—进程层次结构



进程介绍—进程状态



进程介绍—进程示例

```
#include <sys/types.h>  /* 提供类型pid_t的定义,在PC机上与int型
   相同 */
#include  <unistd.h>     /* 提供系统调用的定义 */
main()
{       pid_t pid;          /*此时仅有一个进程*/
         printf("PID before fork():%d\n",(int)getpid());
      pid=fork();
     /*此时已经有两个进程在同时运行*/
    if(pid<0)               printf("error in fork!");
        else if(pid==0)
             printf("I am the child process, my process ID is %d\n",getpid());
      else
             printf("I am the parent process, my process ID is %d\n",getpid());
}
```

进程介绍—进程示例

编译并运行这个程序：

$gcc fork_test.c -o fork_test

$./fork_test

PID before fork()：1991

I am the parent process, my process ID is 1991

I am the child process, my process ID is 1992

再运行一遍，输出结果可能不同。

读者考虑一下为什么？

进程控制块

★对进程进行全面描述的数据结构

Linux中把对进程的描述结构叫做task_struct:

```
struct task_struct {
  …
  …
          }
```

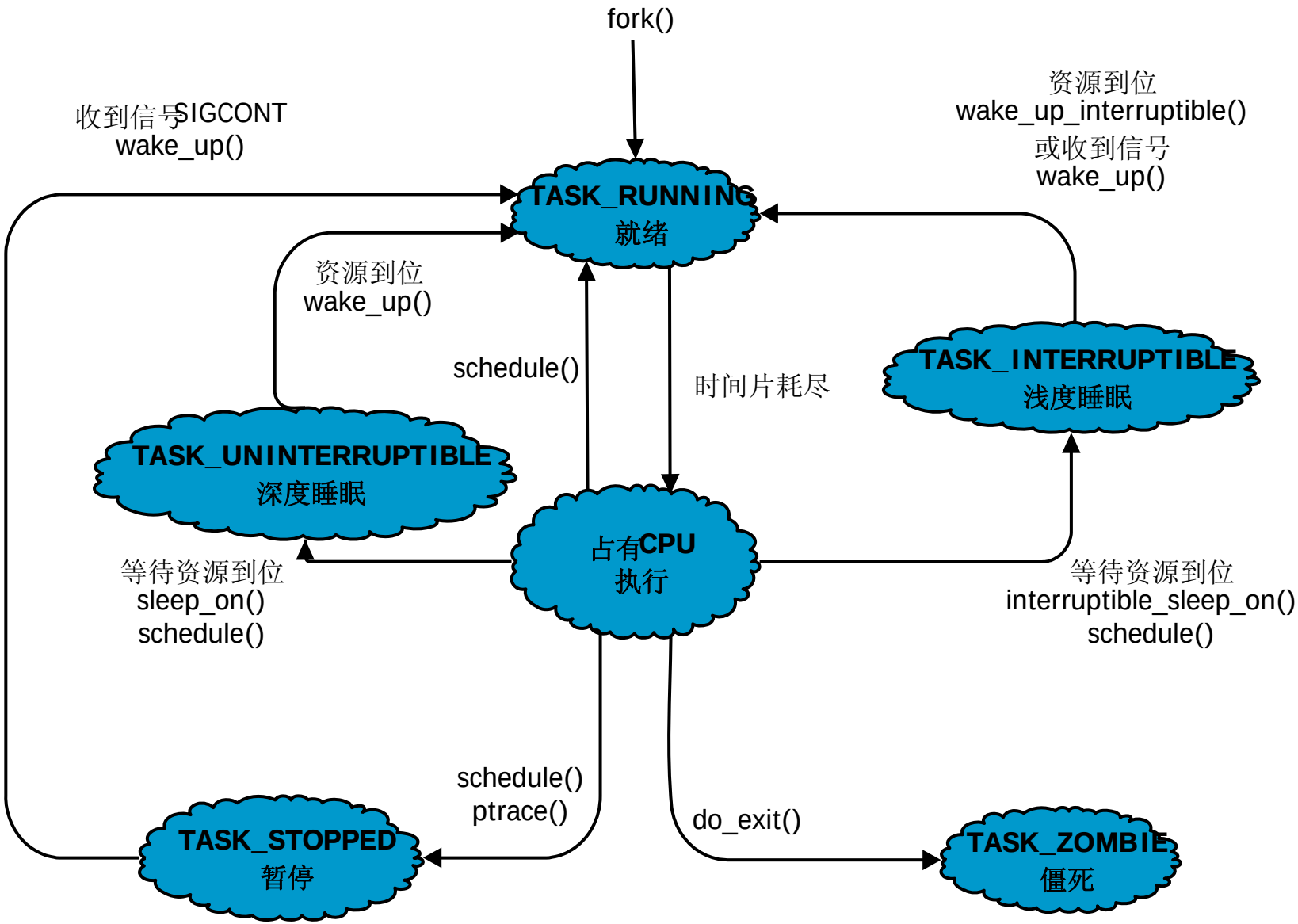
★传统上，这样的数据结构被叫做**进程控制块**

PCB (process control blaock)

进程控制块—信息分类

- ★状态信息—描述进程动态的变化。
- ★链接信息—描述进程的父/子关系。
- ★各种标识符—用简单数字对进程进行标识。
- ★进程间通信信息—描述多个进程在同一任务上协作工作。
- ★时间和定时器信息—描述进程在生存周期内使用CPU时间的统计、计费等信息。
- ★调度信息—描述进程优先级、调度策略等信息。
- ★文件系统信息—对进程使用文件情况进行记录。
- ★虚拟内存信息—描述每个进程拥有的地址空间。
- ★处理器环境信息—描述进程的执行环境(处理器的寄存器及堆栈等)

进程控制块—Linux进程状态及转换



进程控制块—进程标识符

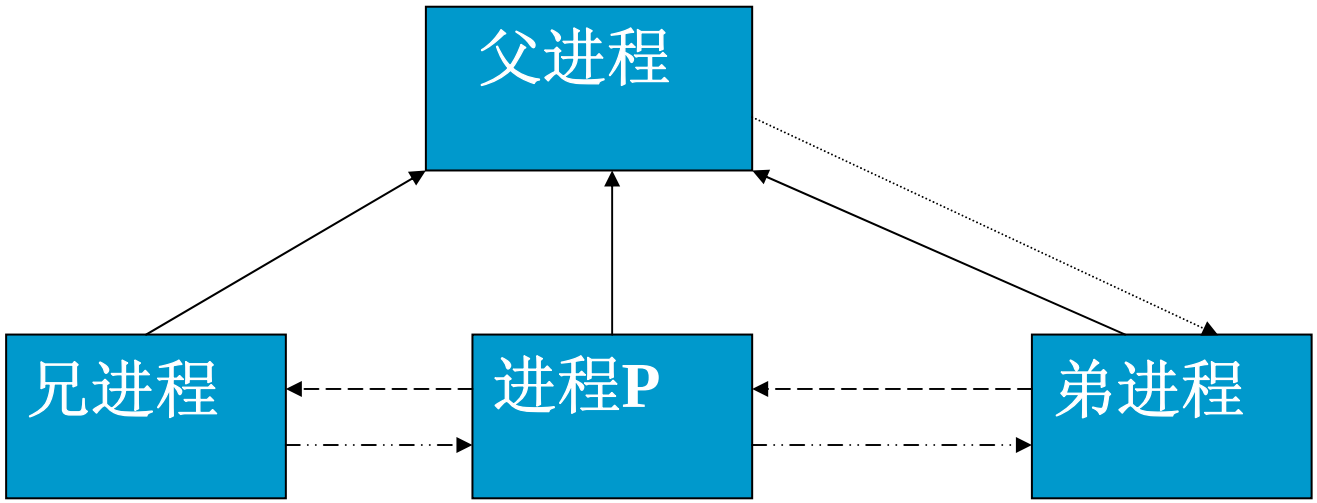
★每个进程都有一个唯一的标识符，内核通过这个标识符来识别不同的进程 。

- 进程标识符PID也是内核提供给用户程序的接口，用户程序通过PID对进程发号施令 。
- PID是32位的无符号整数，它被顺序编号

★每个进程都属于某个用户组。

- task_struct结构中定义有用户标识符UID (User Identifier) 和组标识符GID (Group Identifier)
- 这两种标识符用于系统的安全控制
- 系统通过这两种标识符控制进程对系统中文件和设备的访问。

进程控制块—进程之间的亲属关系



指向父进程　　指向子进程

指向兄进程　　指向弟进程

进程控制块一部分内容的描述

上面通过对进程状态、标识符及亲属关系的描述，我们可以把这些域描述如下：

```
task_struct{
    long state;   /*进程状态*/
    int pid,uid,gid;   /*一些标识符*/
    struct   task_struct      *parent,   *child,
  *o_sibling, *y_sibling   /*一些亲属关系*/
    …
}
```

进程控制块—如何存放

C语言使用下列的联合结构表示这样一个混合结构：

```
union task_union {
   struct task_struct task;
   unsigned long stack[2408];
};
```

Linux调用alloc_task_struct()函数分配8KB的task_union 内存区，调用free_task_struct()函数释放它

进程控制块—如何存放

- C语言使用下列的联合结构表示这样一个混合结构：

```
union task_union {
    struct task_struct task;
    unsigned long stack[2408];
};
```

- Linux调用alloc_task_struct()函数分配8KB的task_union内存区，调用free_task_struct()函数释放它

进程控制块—如何存放

- 把PCB与内核栈放在一起具有以下好处：

(1) 内核可以方便而快速地找到PCB，用伪代码描述如下：

```
p = (struct task_struct *) STACK_POINTER & 0xffffe000
```

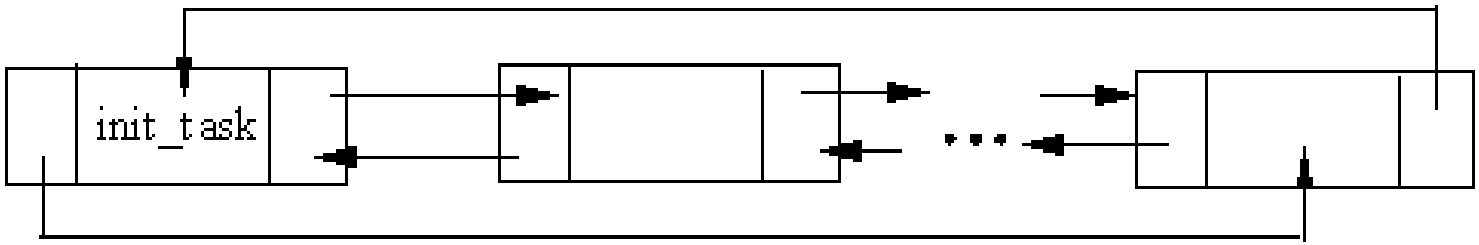
(2) 避免在创建进程时动态分配额外的内存

- 在Linux中，为了表示当前正在运行的进程，定义了一个current宏，可以把它看作全局变量来用，例如current->pid返回正在执行的进程的标识符

进程的组织方式-进程链表

- 在**task_struct**中定义如下：

  task_struct *prev_task, *next_task



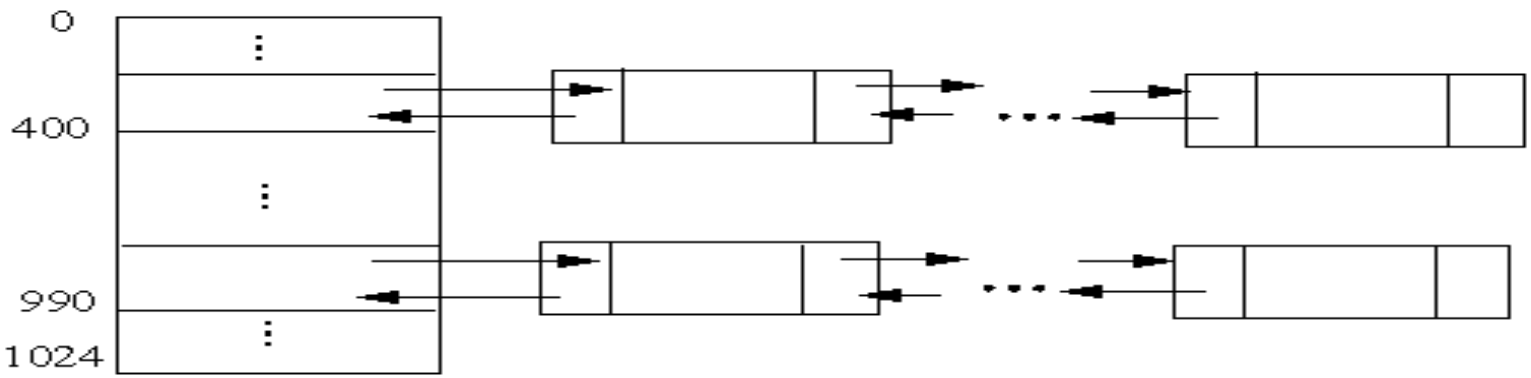
- **宏for_each_task()遍历整个进程链表**

```
#define for_each_task(p) \
        for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

进程的组织方式-哈希表

❖哈希函数

```
#define pid_hashfn(x) \
        ((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))
```



图为地址法处理冲突时的哈希表

❖假定哈希表头为：

```
struct task_struct  *pidhash[PIDHASH_SZ]
```

对给定的PID,如何快速找到对应进程？

进程的组织方式-可运行队列

- 把可运行状态的进程组成一个双向循环链表，也叫可运行队列（runqueue）

- 在task_struct结构中定义了两个指针。

  struct task_struct *next_run, *prev_run;

- init_task起链表头的作用

- 在调度程序运行过程中，允许队列中加入新出现的可运行态进程，新出现的可运行态进程插入到队尾

进程的组织方式-等待队列

- 等待队列表示一组睡眠的进程

- 可以把等待队列定义为如下结构：

```
struct wait_queue {
    struct task_struct * task;
    struct wait_queue * next;
};
```

- 如何让正在运行的进程等待某一特定事件？ Linux内核中实现了sleep_on()函数，请给出该函数的实现。
- 如果要让等待的进程唤醒，就调用唤醒函数wake_up()，它让待唤醒的进程进入TASK_RUNNING状态。

进程调度-调度算法考虑的因素

★公平：保证每个进程得到合理的CPU时间。

★高效：使CPU保持忙碌状态，即总是有进程在CPU上运行。

★响应时间：使交互用户的响应时间尽可能短。

★周转时间：使批处理用户等待输出的时间尽可能短。

★吞吐量：使单位时间内处理的进程数量尽可能多。

进程调度-调度算法

★时间片轮转调度算法

- 系统使每个进程依次地按时间片轮流地执行

★优先权调度算法

- 非抢占式优先权算法
- 抢占式优先权调度算法

★多级反馈队列调度

- 优先权高的进程先运行给定的时间片，相同优先权的进程轮流运行给定的时间片

★实时调度

- 一般采用抢占式调度方式

进程调度-时间片

- ★时间片表明进程在被抢占前所能持续运行的时间。
  - ❖时间片过长会导致系统对交互的响应表现欠佳
  - ❖时间片太短会明显增大进程切换带来的处理器时间。
- ★Linux调度程序提高交互式程序的优先级，让它们运行得更频繁，于是，调度程序提供较长的默认时间片给交互式程序
- ★Linux调度程序还根据进程的优先级动态调整分配给它的时间片

进程调度-调度时机

★进程状态转换的时刻：进程终止、进程睡眠

★ 当前进程的时间片用完时；

★设备驱动程序运行时；

★从内核态返回到用户态时；

进程调度-与调度相关的域

- need_resched：调度标志，以决定是否调用schedule( )函数。
- counter：进程处于可运行状态时所剩余的时钟节拍（即时钟中断的间隔时间，为10ms或1ms）数。这个域也叫动态优先级。
- priority：进程的基本优先级或叫“静态优先级”
- rt_priority：实时进程的优先级
- policy：调度的类型，允许的取值是：
  - SCHED_FIFO：先入先出的实时进程
  - SCHED_RR：时间片轮转的实时进程
  - SCHED_OTHER：普通的分时进程。

进程调度-衡量值得运行的程度

```
static inline int goodness(struct task_struct * p, struct
    task_struct *prev) {
      int weight;      / * 权值，作为衡量进程是否运行的唯一依据 */
    if (p->police!=SCHED_OTHER)  /*实时进程*/
        {weight = 1000 + p->rt_priority;
        goto out }
       weight = p->counter;       /*普通进程*/
if(!weight)    / *p用完了时间片*/
  goto out;
       if(p ==prev)    / * 细微调整 */
weight+= 1;
         weight+=p->priority
out:
   return weight
     }
```

调度函数schedule( )-变量说明

- ★struct task_truct current, *prev, *next;
- ★current:全局变量，表示当前正在运行的进程。
- ★ prev：局部变量，表示调度发生之前运行的进程，用它保存current的值。
- ★ next：局部变量，表示调度发生之后要运行的进程。
- ★c:局部变量，进程值得运行的程度
- ★schedule( )函数的关键操作是设置局部变量next，以使next代替prev而指向被选中进程的PCB。

调度函数schedule( ) 片段

★如果prev进程时间片用完，而且还是实时进程，就给它分配新的时间片，并让它到可运行队列末尾：

```
if (prev->policy == SCHED_RR&&!prev->counter) {
        prev->counter = prev->priority;
    move_last_runqueue(prev);
}
```

调度函数schedule( ) 片段

★如果prev进程处于浅度睡眠状态，而且它有未处理的信号，就应当唤醒它，给它一个被选择执行的机会。

```
if (prev->state == TASK_INTERRUPTIBLE &&
    signal_pending(prev))
     prev->state = TASK_RUNNING;
```

★如果prev进程不是可运行状态，说明它必须等待某一外部资源，那就让它去睡觉，因此，必须从运行队列链表中删除prev:

```
if (prev->state != TASK_RUNNING)
        del_from_runqueue(prev);
```

调度函数schedule( ) 片段

- 把next初始化为要检查的第一个可运行进程。

```
if (prev->state == TASK_RUNNING) {
        next = prev;
      if (prev->policy & SCHED_YIELD) {   /*自愿放弃CPU */
          prev->policy &= ~SCHED_YIELD;
          c = 0;
       } else
           c = goodness(prev, prev);
  } else {
      c = -1000;    /*永不选中，运行队列链表只包含 init_task */
      next = &init_task;
}
```

调度函数schedule( ) 片段

schedule( ) 在可运行进程队列上重复调用**goodness( )**函数以确定最佳后选者：

```
p = init_task.next_run;
while (p != &init_task) {
    weight = goodness(prev, p);
    if (weight > c) {
        c = weight;
        next = p;
    }
    p = p->next_run;
}
```

调度函数schedule( ) 片段

现在到了**schedule( )**的结束部分：如果已选择了一个进程，进程切换必定发生：

```
if (prev != next) {
    kstat.context_swtch++;  / * 统计进程切换的次数*/
    switch_to(prev,next);  /* 从prev切换到next*/
}
return;
```

进程的创建

- ★进程的创建fork()借用现实世界的“克隆”技术
- ★ 子进程克隆父进程，但不仅如此。而是采用了“写时复制”技术
- ★父进程并不是把自己的所有东西马上都给儿子，而是直到儿子真正需要时才给它。
- ★也就是当父进程或子进程试图修改某些内容时，内核才在修改之前将被修改的部分进行拷贝－这叫做写时复制。
- ★fork()的实际开销就是复制父进程的页表以及给子进程创建唯一的PCB

进程的创建－fork()

进程创建函数fork()和线程创建函数clone()都调用内核函数do_fork()，其主要操作：

★调用alloc_task_struct( )函数以获得8KB的union task_union内存区，用来存放新进程的PCB和内核栈。

★ 让当前指针指向父进程的PCB，并把父进程PCB的内容拷贝到刚刚分配的新进程的PCB中。

★检查新创建这个子进程后，当前用户所拥有的进程数目没有超出给他分配的资源的限制。

★现在， do_fork( )已经获得它从父进程能利用的几乎所有的东西；剩下的事情就是集中建立子进程的新资源，并让内核知道这个新进程已经呱呱落地。

进程的创建—fork()

- ★接下来，子进程的状态被设置为TASK_UNINTERRUPTIBLE以保证它不会马上投入运行。
- ★调用get_pid()为新进程获取一个有效的PID。
- ★然后，更新不能从父进程继承的PCB的其他所有域，例如，进程间亲属关系的域。
- ★把新的PCB插入进程链表，以确保进程之间的亲属关系。
- ★把新的PCB插入pidhash哈希表。
- ★把子进程PCB的状态域设置成TASK_RUNNING，并调用wake_up_process( )把子进程插入到运行队列链表。
- ★让父进程和子进程平分剩余的时间片。
- ★返回子进程的PID，这个PID最终由用户态下的父进程读取

线程—独立执行的一个函数

★Linux把线程和进程一视同仁，不过线程本身拥有的资源少，共享进程的资源，如地址空间。

★Linux内核线程— 在内核态下创建、独立执行的一个内核函数：

```
int kernel_thread(int (*fn)(void *), void * arg,
                  unsigned long flags)
{   pid_t p;
    p = clone( 0, flags | CLONE_VM );
    if ( p )        /* 父*/
        return p;
    else {          /* 子*/
        fn(arg);   exit( );
   } }
```

内核线程一周期执行的任务

★内核线程是通过系统调用clone()来实现的，使用CLONE_VM标志说明内核线程与调用它的进程(current)具有相同的进程地址空间.

★ 由于调用进程是在内核中调用kernel_thread()，因此当系统调用返回时，子进程也处于内核态中，而子进程随后调用fn，当fn退出时，子进程调用exit()退出，所以子进程是在内核态运行的.

★由于内核线程是在内核态运行的，因此内核线程可以访问内核中数据，调用内核函数. 运行过程中不能被抢占等等.

★内核线程也可以叫内核任务，它们周期性地执行，例如，磁盘高速缓存的刷新，网络连接的维护，页面的换入换出等等。

几个特殊身份的内核线程

★没事闲逛的0号进程－从无到有诞生的第一个线程，执行cpu_idle()函数（省电又少热）。

★ 既是内核线程也是1号用户进程的*init*。Init进程诞生后就不愿意死亡了，它创建和监控操作系统外层所有进程的活动。

还有另外四个线程：

★kflushd（即bdflush)线程：刷新“脏”缓冲区中的内容到磁盘以归还内存。

★Kupdate线程：刷新旧的“脏”缓冲区中的内容到磁盘以减少文件系统不一致的风险。

★Kpiod线程：把属于共享内存映射的页面交换出去。

★Kswapd线程：执行内存回收功能。

进程的系统调用

★Fork() －父亲克隆一个儿子。执行fork()之后，兵分两路，两个进程并发执行。

★Exec() －新进程脱胎换骨，离家独立，开始了独立工作的职业生涯。

★Wait() －等待不仅仅是阻塞自己，还准备对僵死的子进程进行善后处理。

★Exit() －终止进程，把进程的状态置为“僵死”，并把其所有的子进程都托付给init进程，最后调用schedule()函数，选择一个新的进程运行。

进程的一生

★随着一句fork，一个新进程呱呱落地，但这时它只是老进程的一个克隆。然后，随着exec，新进程脱胎换骨，离家独立，开始了独立工作的职业生涯。

★人有生老病死，进程也一样，它可以是自然死亡，即运行到main函数的最后一个"}"，从容地离我们而去；也可以是中途退场，退场有2种方式，一种是调用exit函数，一种是在main函数内使用return，无论哪一种方式，它都可以留下留言，放在返回值里保留下来；甚至它还可能被谋杀，被其它进程通过另外一些方式结束它的生命。

★进程死掉以后，会留下一个空壳，wait站好最后一班岗，打扫战场，使其最终归于无形。这就是进程完整的一生。

与调度相关的系统调用

系统调用	描述
nice( )	改变一个普通进程的优先级
getpriority( )	取得一组普通进程的最大优先级
setpriority( )	设置一组普通进程的优先级
sched_getscheduler( )	取得一个进程的调度策略
sched_setscheduler( )	设置一个进程的调度策略和优先级
sched_getparam( )	取得一个进程的调度优先级
sched_setparam( )	设置一个进程的优先级
sched_yield( )	不阻塞的情况下自愿放弃处理机
sched_get_ priority_min( )	取得某种策略的最小优先级
sched_get_ priority_max( )	取得某种策略的最大优先级

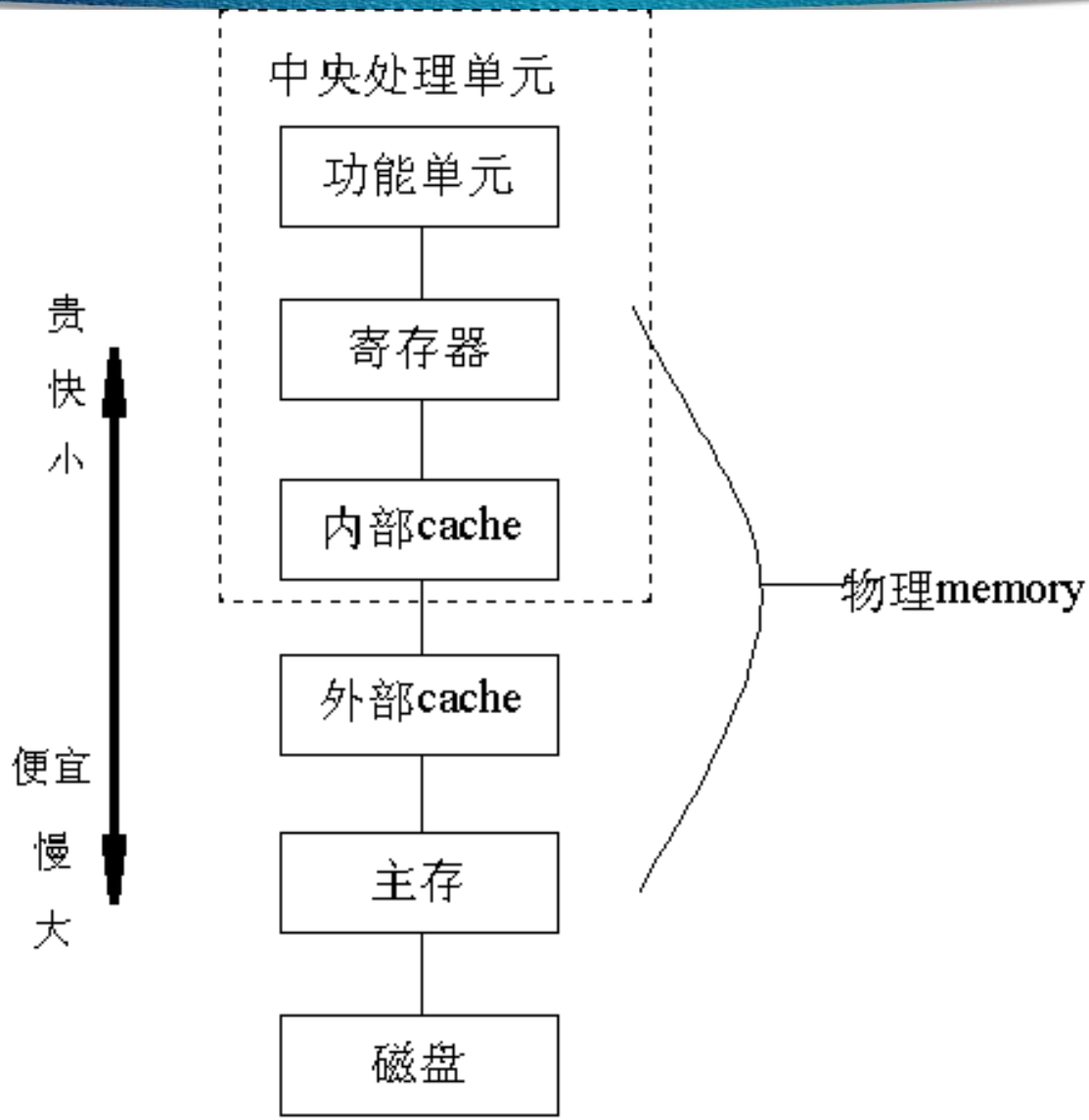
小节

- 进程是一个抽象概念，是对程序执行过程的抽象
- 进程控制块是进程这一抽象概念在计算机中的描述
- 进程的组织采用了树、链表，哈希表，队列等。
- Linux采用了时间片轮转的优先级调度方式
- 进程采用“写时复制”技术创建一个新进程
- 进程从诞生到死亡涉及四种系统调用
- 程序开发人员可以改变进程的优先级，甚至调度策略

第四章 内存管理

- Linux的内存管理
- 进程的用户空间管理
- 请页机制
- 物理内存的分配与回收
- 交换机制
- 内存管理示例

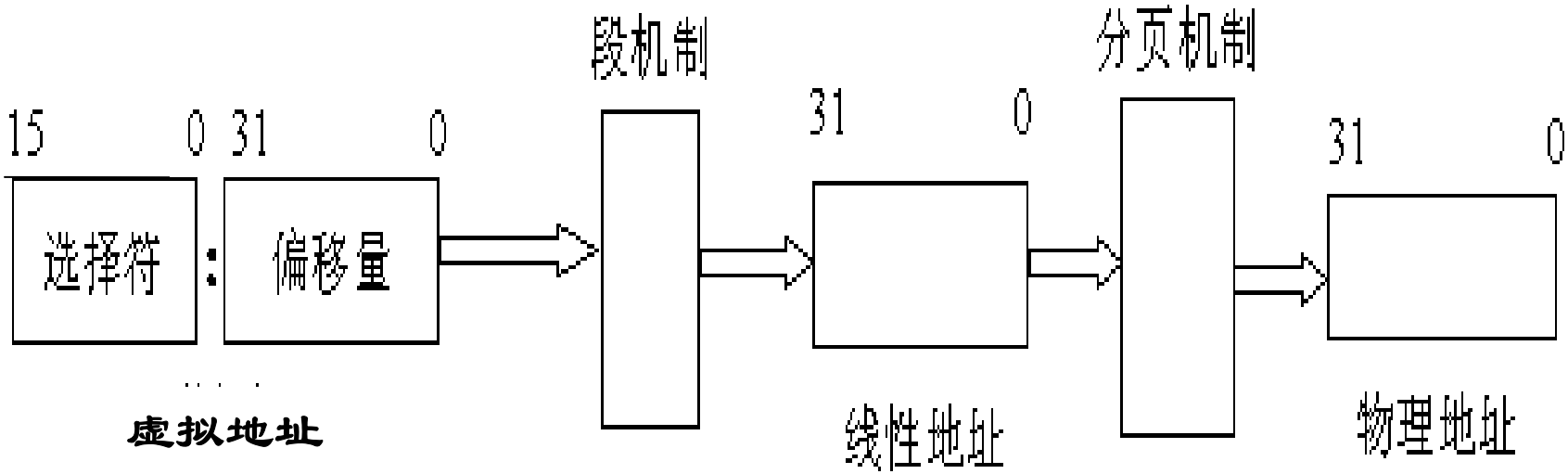
内存的层次结构



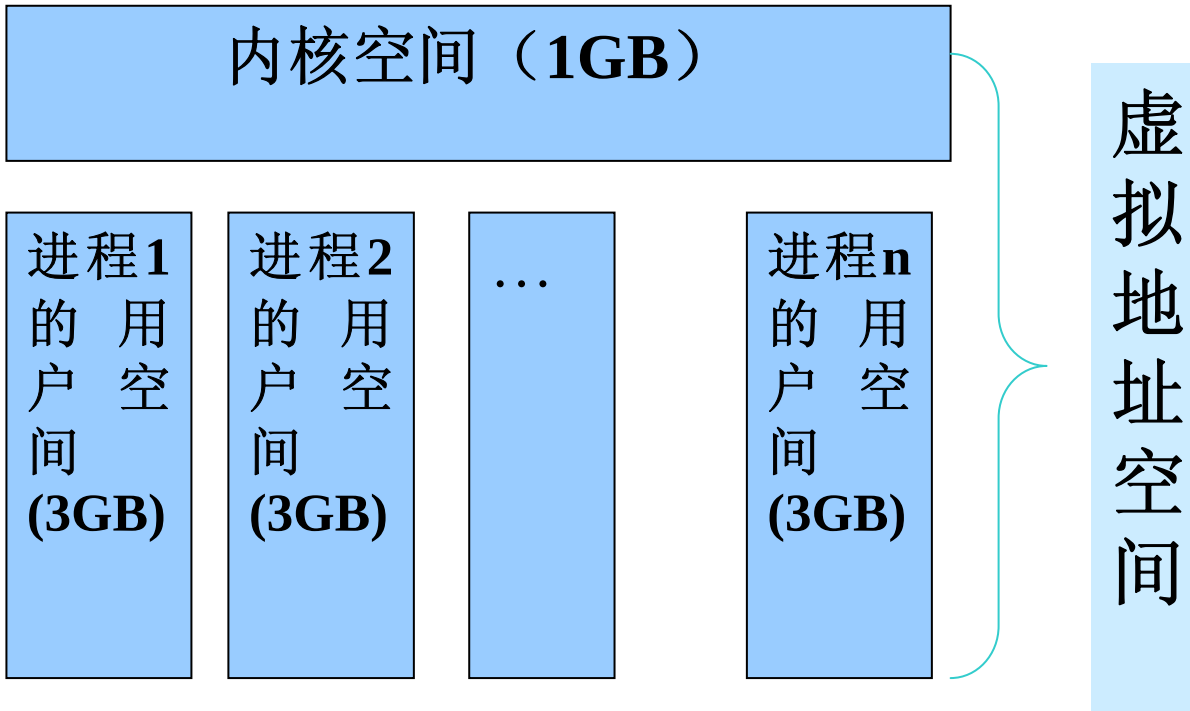
扩大了的记忆—虚拟内存

- 虚拟内存的**基本思想**：在计算机中运行的程序，其代码、数据和堆栈的总量可以超过实际内存的大小，操作系统只将当前使用的程序块保留在内存中，其余的程序块则保留在磁盘上。必要时，操作系统负责在磁盘和内存之间交换程序块。

虚地址到实地址转换



虚拟内存、内核空间和用户空间



虚拟内存、内核空间和用户空间

- 虚拟内存－共4G字节，分为内核空间（最高的1G字节）和用户空间（较低的3G字节）两部分，每个进程最大拥有3G字节私有虚存空间
- 地址转换－通过页表把虚存空间的一个地址转换为物理空间中的实际地址。

内核空间到物理内存的映射

- 内核空间由所有进程共享，其中存放的是内核代码和数据，即“内核映象”
- 进程的用户空间中存放的是用户程序的代码和数据
- 内核空间映射到物理内存总是从最低地址（0x00000000）开始，使之在内核空间与物理内存之间建立简单的线性映射关系。

内核空间到物理内存的映射

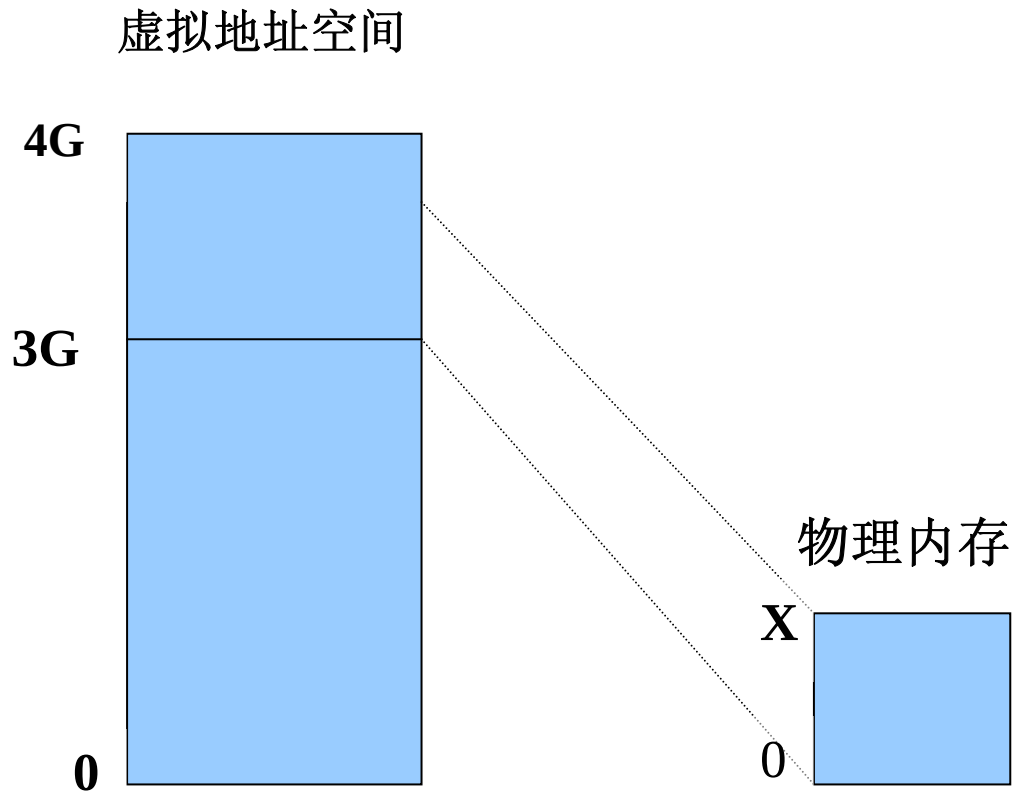


图4.1 内核的虚拟地址空间到物理地址空间的映射

虚拟内存实现机制

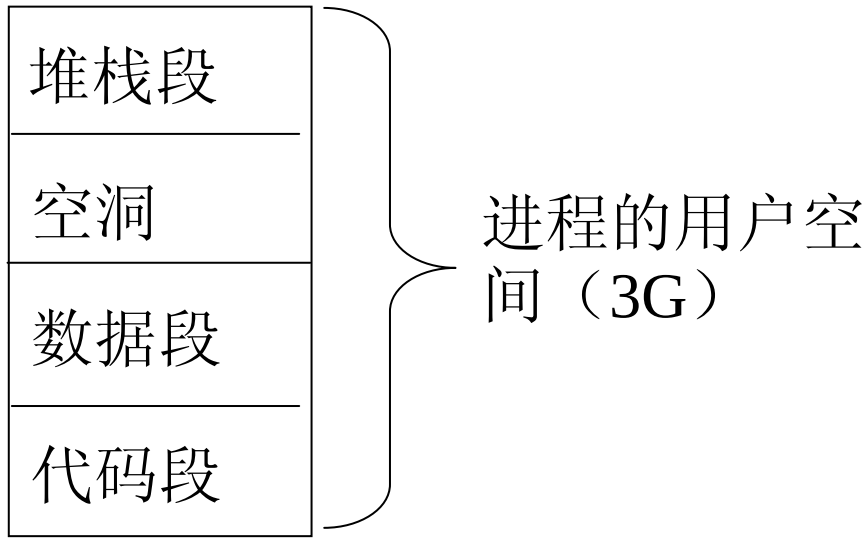
- Linux虚拟内存的实现需要多种机制的支持
  - 地址映射机制
  - 请页机制
  - 内存分配和回收机制
  - 交换机制
  - 缓存和刷新机制

虚拟内存实现机制及之间的关系

图4.2　虚拟内存实现机制及之间的关系

进程的用户空间管理

- 每个进程经编译、链接后形成的二进制映像文件有一个代码段和数据段
- 进程运行时须有独占的堆栈空间



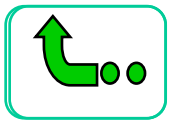
进程用户空间

- Linux把进程的用户空间划分为一个个区间，便于管理
- 一个进程的用户地址空间主要由mm_struct结构和vm_area_structs结构来描述。
- mm_struct结构它对进程整个用户空间进行描述
- vm_area_structs结构对用户空间中各个区间(简称虚存区)进行描述

mm_struct 结构

```
struct mm_struct {
        atomic_t count;
        pgd_t * pgd;
        int map_count;
        struct semaphore mmap_sem;
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
        unsigned long rss, total_vm, locked_vm;
        unsigned long def_flags;
        struct vm_area_struct *mmap, *mmap_avl, *mmap_cache;
        unsigned long swap_cnt;
        unsigned long swap_address;
};
```

域名	说　　明
count	对mm_struct结构的引用进行计数。为了在Linux中实现线程，内核调用clone派生一个线程，线程和调用进程共享用户空间，即mm_struct结构，派生后系统会累加mm_struct中的引用计数。
pgd	进程的页目录基地址，当调度程序调度一个进程运行时，就将这个地址转成物理地址，并写入控制寄存器（CR3）
map_count	在进程的整个用户空间中虚存区的个数
semaphore	对mm_struct结构进行串行访问所使用的信号量
Start_code,end_code, start_data, end_data	进程的代码段和数据段的起始地址和终止地址
start_brk, brk, start_stack;	每个进程都有一个特殊的地址区间，这个区间就是所谓的**堆**，也就是图4.4中的空洞。前两个域分别描述堆的起始地址和终止的地址，最后一个域描述堆栈段的起始地址。
arg_start, arg_end, env_start, env_end	命令行参数所在的堆栈部分的起始地址和终止地址； 环境串所在的堆栈部分的起始地址和终止地址
rss, total_vm, locked_vm	进程贮留在物理内存中的页面数，进程所需的总页数，被锁定在物理内存中的页数。
mmap	vm_area_struct虚存区结构形成一个单链表，其基址由小到大排列
mmap_avl	vm_area_struct虚存区结构形成一个颗AVL平衡树
mmap_cache	最近一次用到的虚存区很可能下一次还要用到，因此，把最近用到的虚存区结构放入高速缓存，这个虚存区就由mmap_cache指向。



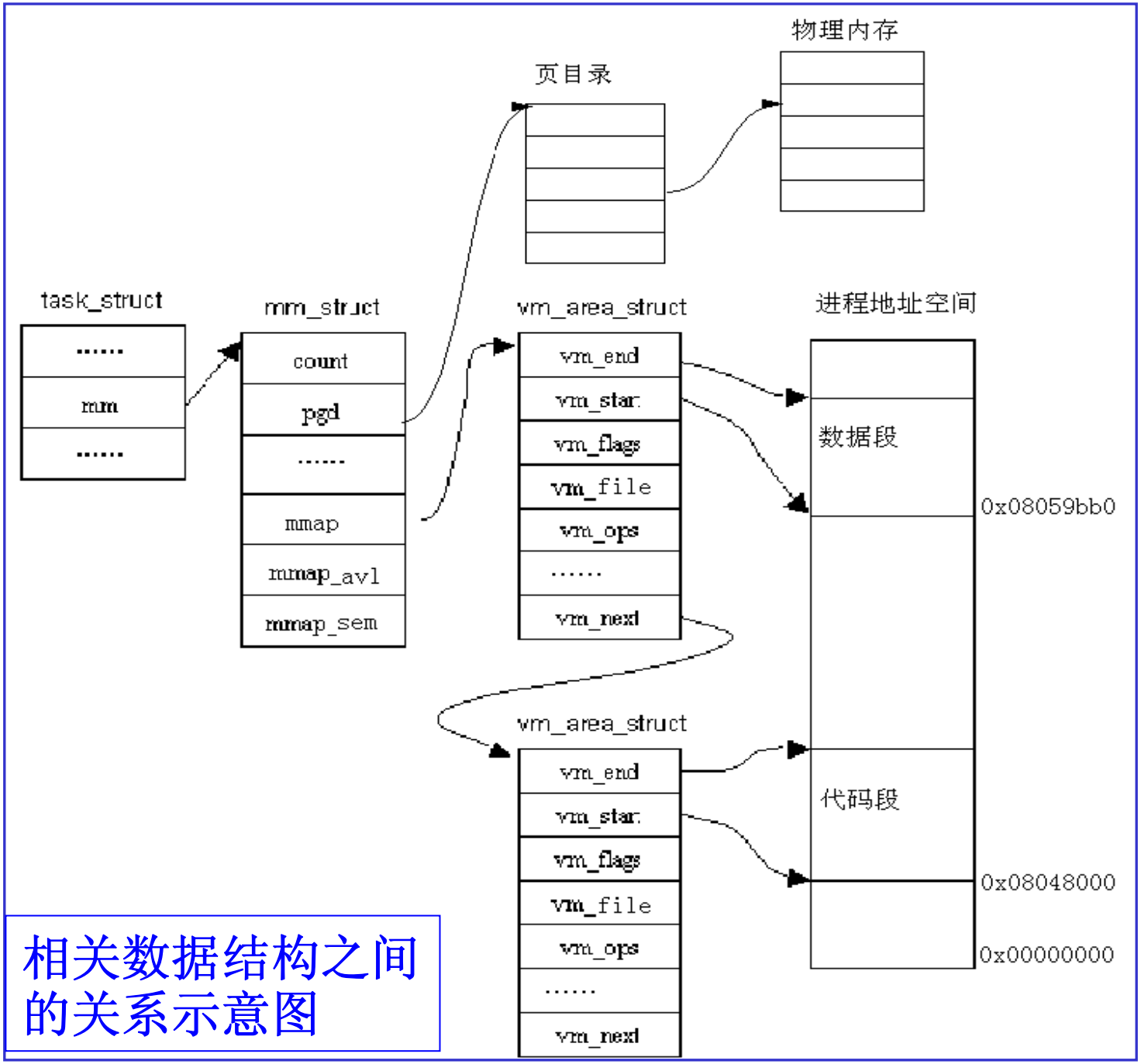
VM_AREA_STRUCT 结构

```
struct vm_area_struct {
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;
        pgprot_t vm_page_prot;
        unsigned short vm_flags;
        struct vm_area_struct *vm_next;
        short vm_avl_height;
        struct vm_area_struct *vm_avl_left, *vm_avl_right;
      struct vm_operations_struct * vm_ops;
       struct vm_area_struct *vm_next_share, **vm_pprev_share;
        unsigned long vm_offset;
        struct file * vm_file;
        unsigned long vm_pte;
   };
```

域名	说　　明
Vm_mm	指向虚存区所在的mm_struct结构的指针。
Vm_start，vm_end	虚存区的起始地址和终止地址。
Vm_page_prot	虚存区的保护权限。
Vm_flags	虚存区的标志。
Vm_next	构成线性链表的指针，按虚存区基址从小到大排列。
vm_avl_height， vm_avl_left，vm_avl_right	这3个域在一起构成AVL树，其中vm_avl_height是该节点距根节点的高度， vm_avl_left和vm_avl_right分别是该节点的左右两个子树。
Vm_ops	对虚存区进行操作的函数。这些给出了可以对虚存区中的页所进行的操作。

相关数据结构间的关系

- 进程控制块是内核中的核心数据结构。
- 在进程的 task_struct 结构中包含一个mm域，它是指向 mm_struct 结构的指针。
- 而进程的 mm_struct结构则包含进程的可执行映像信息以及进程的页目录指针pgd等。
- 该结构还包含有指向 vm_area_struct 结构的几个指针，每个 vm_area_struct 代表进程的一个虚拟地址区间。



相关数据结构之间的关系示意图

创建进程用户空间

- fork()系统调用在创建新进程时也为该进程创建完整的用户空间
- 具体而言，是通过拷贝或共享父进程的用户空间来实现的，即内核调用copy_mm( )函数，为新进程建立所有页表和mm_struct结构
- Linux利用“写时复制”技术来快速创建进程

虚存映射

- 执行一个进程时，其可执行映像必须装入进程的用户地址空间
- 虚存映射：即把文件从磁盘映射到进程的用户空间，对文件的访问转化为对虚存区的访问
- 有共享的、私有的虚存映射和匿名映射
- 当可执行映像映射到进程的用户空间时，将产生一组 vm_area_struct 结构来描述各虚拟区间的起始点和终止点

进程的虚存区举例

例：exam.c

```
int main()
{
printf(“virtual area test!”);
}
```

地址范围	许可权	偏移量	所映射的文件
08048000-08049000	r-xp	00000000	/home/test/exam
08049000-0804a000	rw-p	00001000	/home/test/exam
40000000-40015000	r-xp	00000000	/lib/ld-2.3.2.so
40015000-40016000	rw-p	00015000	/lib/ld-2.3.2.so
40016000-40017000	rw-p	00000000	匿名
4002a000-40159000	r-xp	00000000	/lib/libc-2.3.2.so
40159000-4015e000	rw-p	0012f000	/lib/libc-2.3.2.so
4015e000-40160000	rw-p	00000000	匿名
bfffe000-c0000000	rwxp	fffff000	匿名

exam进程的虚存区

与用户空间相关的主要系统调用

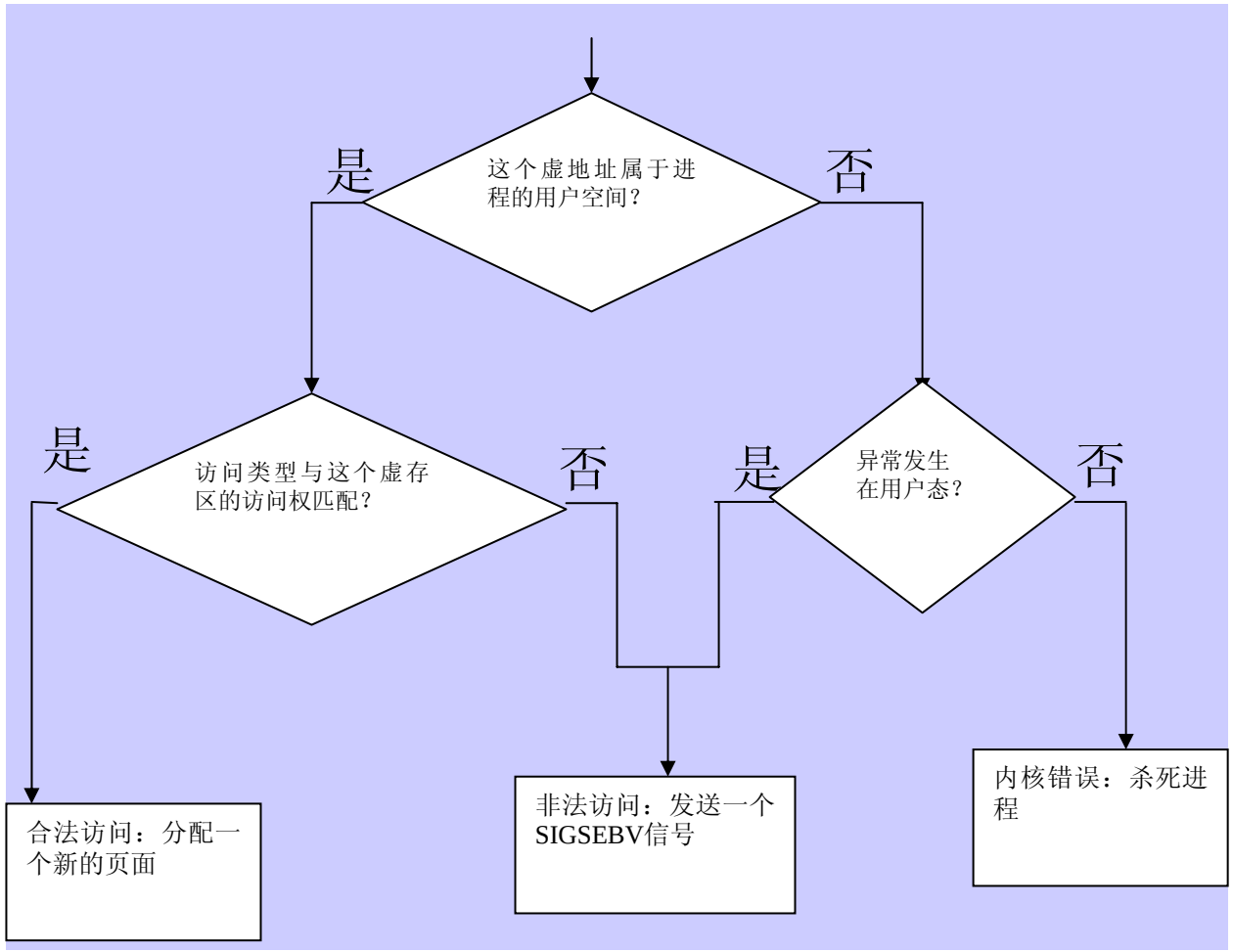
系统调用	描述
fork()	创建具有新的用户空间的进程,用户空间中的所有页被标记为“写时复制”，且由父子进程共享，当其中的一个进程所访问的页不在内存时，这个页就被复制一份。
mmap()	在进程的用户空间内创建一个新的虚存区。
munmap()	销毁一个完整的虚存区或其中的一部分，如果要取消的虚存区位于某个虚存区的中间，则这个虚存区被划分为两个虚存区。
exec()	装入新的可执行文件以代替当前用户空间。
Exit()	销毁进程的用户空间及其所有的虚存区。

请页机制－实现虚存管理的重要手段

- 进程运行时，CPU访问的是用户空间的虚地址
- Linux仅把当前要使用的少量页面装入内存，需要时再通过请页机制将特定的页面调入内存
- 当要访问的虚页不在内存时，产生一个页故障并报告故障原因

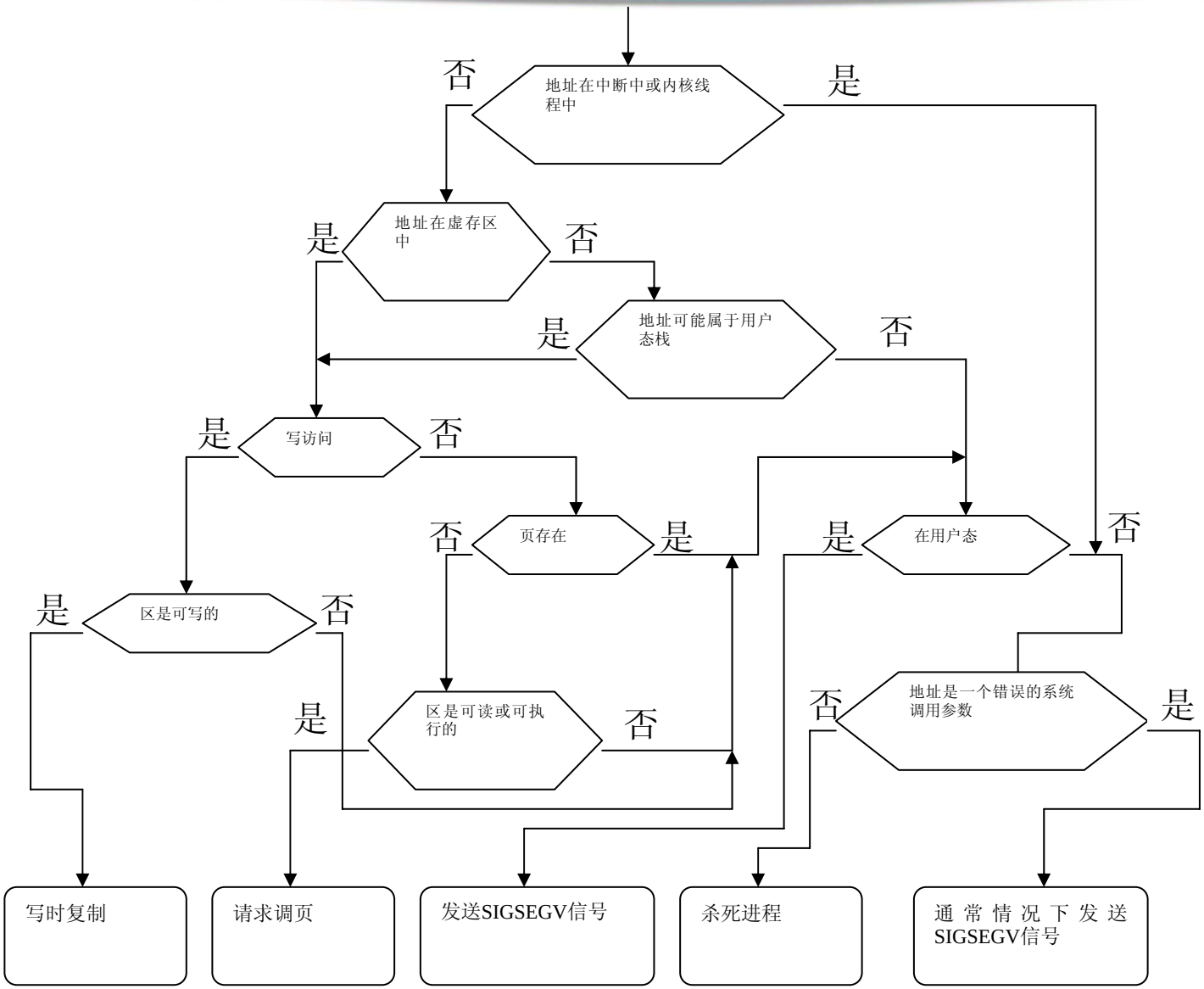
缺页异常处理程序

总体方案



< >

缺页异常处理流程图


地址在中断中或内核线程中
否
是
地址在虚存区中
是
否
地址可能属于用户态栈
是
否
写访问
是
否
页存在
否
是
在用户态
是
否
区是可写的
是
否
区是可读或可执行的
是
否
地址是一个错误的系统调用参数
否
是
写时复制
请求调页
发送SIGSEGV信号
杀死进程
通常情况下发送SIGSEGV信号

请求调页－动态内存分配技术

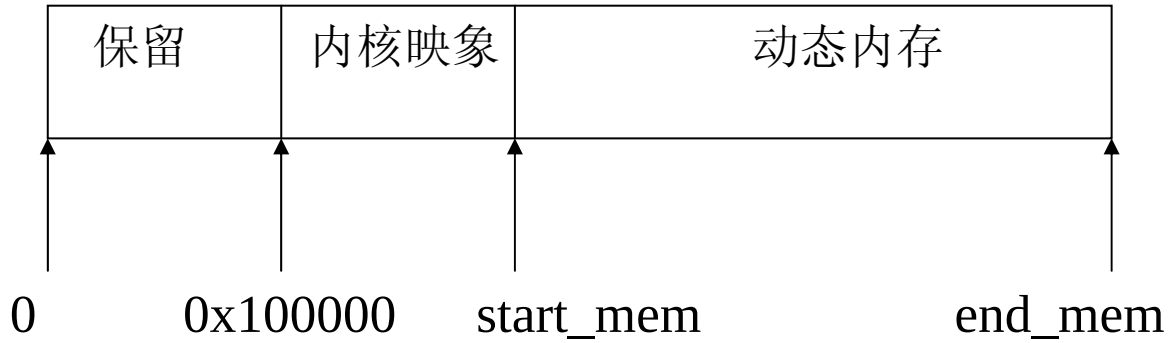
- 请求调页：把页面的分配推迟到进程要访问的页不在物理内存时为止，由此引起一个缺页异常
- 引入原因：进程开始运行时并不访问其地址空间中的全部地址
- 程序的局部性原理保证请求调页从总体上使系统有更大的吞吐量。

写时复制（copy－on－write）技术

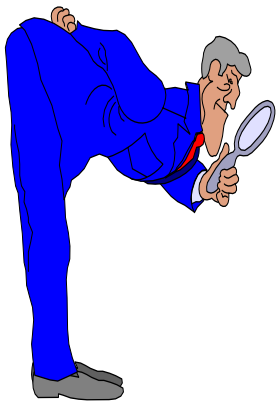
- 写时复制技术可以推迟、甚至免除数据的拷贝
- 进程创建之初内核并不复制整个进程空间，而是使父子进程以只读方式共享同一个拷贝
- 数据只有在需要写入时才会被复制，从而使各个进程拥有各自的拷贝

物理内存的分配与回收

- 在Linux中，CPU所访问的地址是虚拟地址空间的虚地址；
- 管理内存页面时，先在虚存空间中分配一个虚存区间，然后才根据需要为此区间分配相应的物理页面并建立起映射
- Linux采用著名的伙伴（Buddy）算法来解决外碎片问题



页面分配与回收算法－伙伴算法



- **Linux**的伙伴算法把所有的空闲页面分为**10**个块链表，每个链表中的一个块含有**2**的幂次个页面（叫做“页块”或简称“块”）
- 大小相同、物理地址连续的两个页块被称为“伙伴”
- 工作原理：首先在大小满足要求的块链表中查找是否有空闲块，若有则直接分配，否则在更大的块中查找。其逆过程就是块的释放，此时会把满足伙伴关系的块合并

物理页面的分配

- 函数__get_free_pages 用于分配物理页块
- 该函数所做的工作如下：
  - 检查所请求的页块大小是否能够被满足
  - 检查系统中空闲物理页的总数是否已低于允许的下界
  - 正常分配。从free_area数组的第order项开始，这是一个mem_map_t链表。
  - 换页。通过下列语句调用函数try_to_free_pages（），启动换页进程

1）如果该链表中有满足要求的页块，则：

将其从链表中摘下；将free_area数组的位图中该页块所对应的位取反，表示页块已用；修改全局变量nr_free_pages（减去分配出去的页数）；根据该页块在mem_map数组中的位置，算出其起始物理地址，返回。

2）如果该链表中没有满足要求的页块，则在free_area数组中顺序向上查找。其结果有二：

a）整个free_area数组中都没有满足要求的页块，此次无法分配，返回。

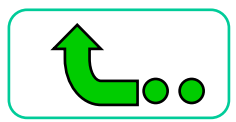
b）找到一个满足要求的页块，则：

将其从链表中摘下； 将free_area数组的位图中该页块所对应的位取反，表示页块已用；修改全局变量nr_free_pages（减去分配出去的页数）；因为页块比申请的页块要大，所以要将它分成适当大小的块。因为所有的页块都由2的幂次的页数组成，所以这个分割的过程比较简单，只需要将它平分就可以：

I. 将其平分为两个伙伴，将小伙伴加入free_area数组中相应的链表，修改位图中相应的位；

II. 如果大伙伴仍比申请的页块大，则转I，继续划分；

III. 大伙伴的大小正是所要的大小，修改位图中相应的位，根据其在mem_map数组中的位置，算出它的起始物理地址，返回。



物理页面的回收

- 函数free_pages用于页块的回收
- 该函数所做的工作如下：
  - 根据页块的首地址addr算出该页块的第一页在mem_map数组的索引；
  - 如果该页是保留的（内核在使用），则不允许回收；
  - 将页块第一页对应的mem_map_t结构中的count域减1，表示引用该页的进程数减了1个。若count域的值不为0，有别的进程在使用该页块，不能回收，仅简单返回
  - 清除页块第一页对应的mem_map_t结构中flags域的PG_referenced位，表示该页块不再被引用；
  - 将全局变量nr_free_pages的值加上回收的物理页数
  - 将页块加入到数组free_area的相应链表中

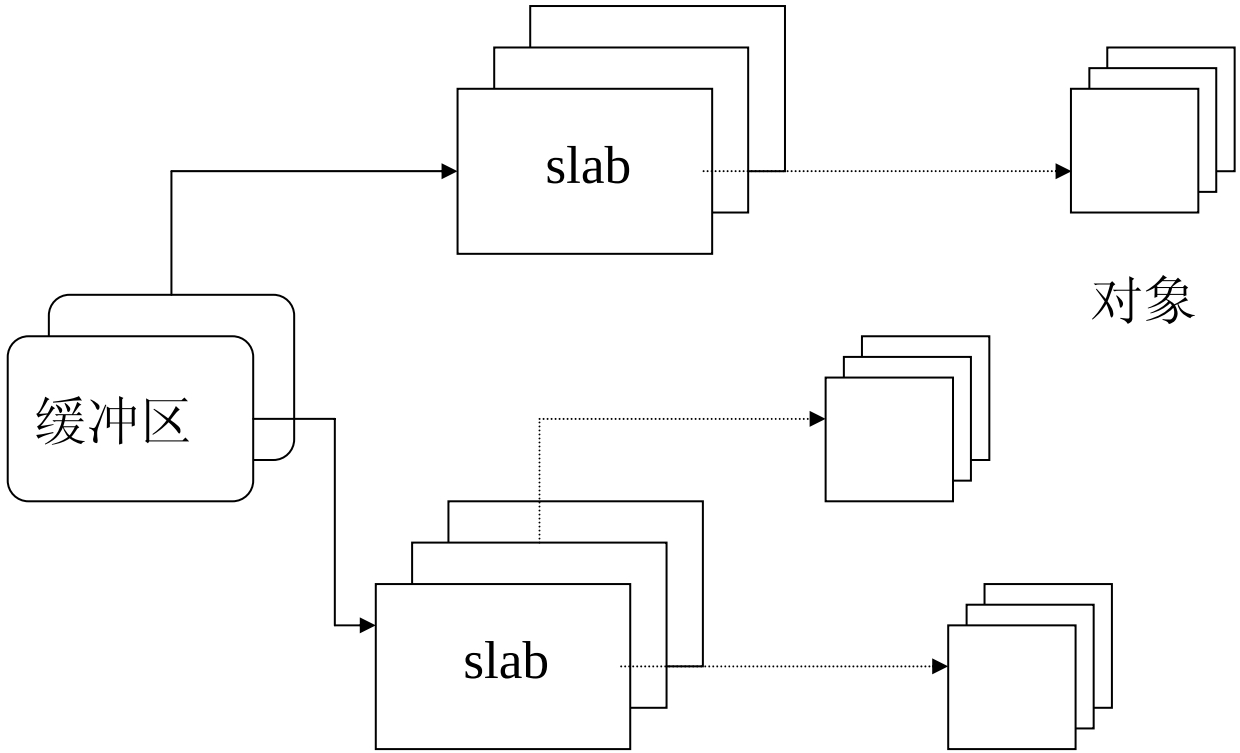
Slab 分配机制－分配小内存

- **Slab**机制提出的原因：
  - 为了减少对伙伴算法的调用次数
  - 内核经常反复使用某一内存区
  - 内存区可根据其使用频率来分类
  - 硬件高速缓存的使用，为尽量减少对伙伴算法的调用提供了另一个理由

Slab 分配机制－分配小内存

- **Slab**分配模式把对象分组放进缓冲区
- **Slab**缓冲区由一连串的“大块（**Slab**）”构成，每个大块中包含若干个同种类型的对象，这些对象或已被分配，或空闲
- 简言之，缓冲区就是主存中的一片区域，把这片区域划分为多个块，每块就是一个**Slab**，每个**Slab**由一个或多个页面组成，每个**Slab**中存放的就是对象

Slab 的组成



Slab专用缓冲区的建立和释放

- 专用缓冲区主要用于频繁使用的数据结构
- 缓冲区是用**kmem_cache_t**类型描述的，通过**kmem_cache_create**（）来建立
- 函数**kmem_cache_create**（）所创建的缓冲区中还没有包含任何**Slab**，因此，也没有空闲的对象。只有以下两个条件都为真时，才给缓冲区分配**Slab**：
  - 已发出一个分配新对象的请求；
  - 缓冲区不包含任何空闲对象；

Slab专用缓冲区的建立和释放

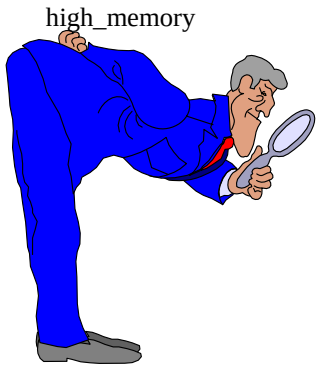
- 创建缓冲区后，可通过函数**kmem_cache_alloc**（）从中获取对象
- 该函数从给定的缓冲区中返回一个指向对象的指针。如果缓冲区中所有的**slab**中都没有空闲的对象，则 **slab**必须调用**__get_free_pages()**获取新的页面
- 使用函数**kmem_cache_free**（）可以释放一个对象，并把它返回给原先的slab

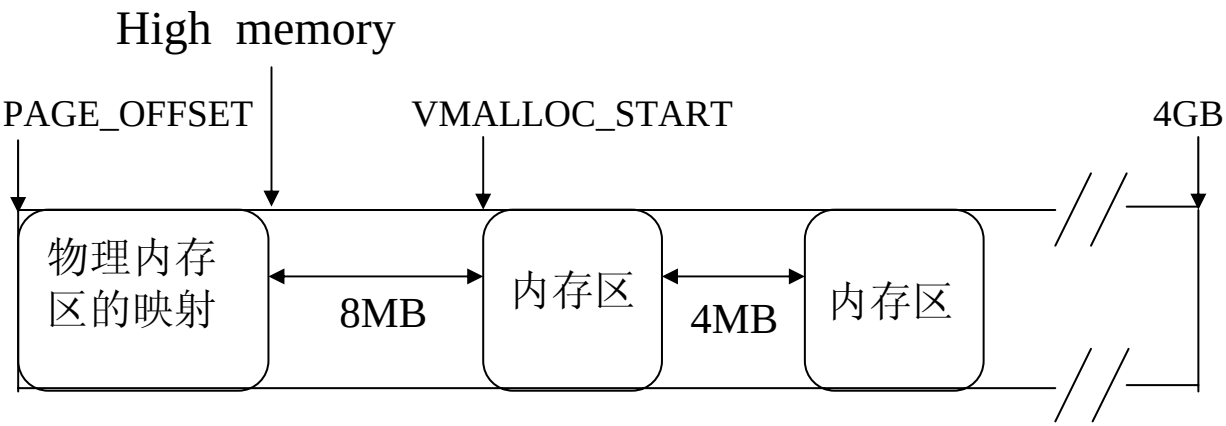
通用缓冲区

- 在内核中初始化开销不大的数据结构可以合用一个通用的缓冲区。
- 通用缓冲区类似于物理页面分配中的大小分区
- 对通用缓冲区的管理采用**Slab**方式
- 当一个数据结构的使用不频繁、或其大小不足一个页面时，没有必要给其分配专用缓冲区，可调用函数**kmallo()** 分配通用缓冲区

内核空间非连续内存区的分配

- 非连续内存处于3G到4G之间的内核空间
- **PAGE_OFFSET为3GB，high_memory为保存物理地址最高值的变量，VMALLOC_START为非连续区的起始地址**





从PAGE_OFFSET开始的内核地址区间

vmalloc()与 kmalloc()之区别

- vmalloc()与 kmalloc()都可用于分配内存
- kmalloc()分配的内存处于3GB～high_memory之间，这段内核空间与物理内存的映射一一对应，而vmalloc()分配的内存在VMALLOC_START～4GB之间，这段非连续内存区映射到物理内存也可能是非连续的
- vmalloc() 分配的物理地址无需连续，而kmalloc() 确保页在物理上是连续的

交换机制

- 当物理内存不足时，**Linux**通过某种机制选出内存中的某些页面换到磁盘上，以便留出空闲区来调入需要使用的页面
- 交换的基本原理：当空闲内存数量小于一个固定的极限值时，就执行换出操作（包括把进程的整个地址空间拷贝到磁盘上）。反之，当调度算法选择一个进程运行时，整个进程又被从磁盘中交换进来

页面交换

- 在**Linux**中，进行交换的单位是页面而不是进程
- 在页面交换中，页面置换算法是影响交换性能的关键性指标，其复杂性主要与换出有关：
  - 哪种页面要换出
  - 如何在交换区中存放页面
  - 如何选择被交换出的页面

选择被换出的页面

- 只有与用户空间建立了映射关系的物理页面才会被换出，内核空间中内核所占的页面则常驻内存
- 进程映像所占的页面，其代码段、数据段可被换入换出，但堆栈段一般不换出
- 通过系统调用**mmap()**把文件内容映射到用户空间时，页面所使用的交换区就是被映射的文件本身
- 进程间共享内存区其页面的换入换出比较复杂
- 映射到内核空间中的页面都不会被换出
- 内核在执行过程中使用的页面要经过动态分配，但永驻内存

在交换区中存放页面

- 交换区也被划分为块，每个块的大小恰好等于一页，一块叫做一个页插槽
- 换出时，内核尽可能把换出的页放在相邻的插槽中，从而减少访问交换区时磁盘的寻道时间
- 若系统使用了多个交换区，快速交换区可以获得比较高的优先级
- 当查找一个空闲插槽时，要从优先级最高的交换区中开始搜索
- 如果优先级最高的交换区不止一个，应该循环选择相同优先级的交换区

页面交换策略

- 策略1：需要时才交换
- 策略2：系统空闲时交换
- 策略3：换出但并不立即释放
- 策略4：把页面换出推迟到不能再推迟为止
- 页面换入／换出及回收的基本思想

< >

- 释放页面。如果一个页面变为空闲可用，就把该页面的page结构链入某个空闲队列free_area，同时页面的使用计数count减1。
- 分配页面。调用___get_free_page()从某个空闲队列分配内存页面，并将其页面的使用计数count置为1。
- 活跃状态。已分配的页面处于活跃状态，该页面的数据结构page通过其队列头结构lru链入活跃页面队列active_list，并且在进程地址空间中至少有一个页与该页面之间建立了映射关系。
- 不活跃“脏”状态。处于该状态的页面其page结构通过其队列头结构lru链入不活跃“脏”页面队列inactive_dirty_list，并且原则是任何进程的页面表项不再指向该页面，也就是说，断开页面的映射，同时把页面的使用计数count减1。
- 将不活跃“脏”页面的内容写入交换区，并将该页面的page结构从不活跃“脏”页面队列inactive_dirty_list转移到不活跃“干净”页面队列，准备被回收。
- 不活跃“干净”状态。页面page结构通过其队列头结构lru链入某个不活跃“干净”页面队列。
- 如果在转入不活跃状态以后的一段时间内，页面又受到访问，则又转入活跃状态并恢复映射。
- 当需要时，就从“干净”页面队列中回收页面，也就是说或者把页面链入到空闲队列，或者直接进行分配。

页面交换守护进程kswapd

- **Linux**内核利用守护进程**kswapd**定期地检查系统内的空闲页面数是否小于预定义的极限，一旦发现空闲页面数太少，就预先将若干页面换出
- **kswapd**相当于一个进程，它有自己的进程控制块**task_struct**结构，与其它进程一样受内核调度，但没有独立的地址空间

内存管理实例

- 希望通过访问用户空间的内存达到读取内核数据的目的，这样便可进行内核空间到用户空间的大规模信息传送，从而应用于高速数据采集等性能要求高的场合
- 从用户空间直接读取内核数据，即利用内存映射功能，将内核中的一部分虚拟内存映射到用户空间，使得访问用户空间地址等同于访问被映射的内核空间地址，从而不再需要数据拷贝操作

相关背景知识

- 在内核空间中调用kmalloc()分配连续物理空间，而调用vmalloc()分配非物理连续空间。
- 我们把kmalloc()所分配内核空间中的地址称为**内核逻辑地址**
- 把vmalloc()分配的内核空间中的地址称为**内核虚拟地址**
- vmalloc()在分配过程中须更新内核页表

代码体系结构介绍

- 跨空间的地址映射主要包括：

  - 找到内核地址对应的物理地址，这是为了将用户页表项直接指向物理地址；
  - 建立新的用户页表项

用户虚存区映射到VK对应的物理内存

用户空间

内核空间

用户虚存区虚拟空间Vm_area_struct

Vk空间vm_struct

用户页表

内核页表

Vma->nopage

物理内存

< >

任务的执行路径

mmap系统调用

File->mma
(mapdrv_mmap)

Vma-
>nopage(map_nopage)

获得vm_area对应的内核逻辑地址
(vaddr_to_kaddr)

得到内核逻辑地址对应的进程页框
(virt_to_page)

获得vm_area对应的内核页表项
(pgd_offset_k
pmd_offset
pte_offset)

获得vm_area对应内核页表项指针
(pte_offset)

获vm_area对应的内核逻辑地址
(page_address)

< >

STEP BY STEP

- 编译map_driver.c为map_driver.o模块,具体参数见Makefile
- 加载模块： insmod map_driver.o
- 生成对应的设备文件
  - 在/proc/devices下找到map_driver对应的设备命和设备号：grep mapdrv /proc/devices
  - 建立设备文件mknod mapfile c 254 0
- 利用maptest读取mapfile文件，将取自内核的信息（”ok”——我们在内核中在vmalloc分配的空间中填放的信息）打印到用户屏幕。

“内核之旅”网站

- http://www.kerneltravel.net/

- 电子杂志栏目是关于内核研究和学习的资料
- 第五期“Linux内存管理”，从应用程序开发者的角度审视Linux的进程内存管理，在此基础上逐步深入到内核中讨论系统物理内存管理和内核内存的使用方法。力求从外到内、水到渠成地引导网友分析Linux的内存管理与使用。
- 下载实例代码进行调试

第五章 中断与异常

- 中断的基本知识
- 中断描述符表的初始化
- 中断处理
- 中断的下半部处理机制
- 中断的应用－时钟中断

中断掠影

- 中断控制的主要优点：
  - CPU只有在I/O需要服务时才响应
- 外部中断：
  - 外部设备所发出的I/O请求
- 内部中断：
  - 也称之为“异常”，是为解决机器运行时所出现的某些随机事件及编程方便而出现的

中断常识

- 中断向量：
  - 中断源的编号
- 外设可屏蔽中断：
  - 屏蔽外部I/O请求
- 异常及非屏蔽中断：
  - CPU内部中断或计算机内部硬件出错引起的异常
- 中断描述符表：
  - 描述中断的相关信息
- 中断相关的汇编指令：

中断向量－中断源的类型

- 中断向量－每个中断源都被分配一个8位无符号整数作为类型码，即中断向量
- 中断的种类：
  - 中断：
    - 外部可屏蔽中断
    - 外部非屏蔽中断
  - 异常：不使用中断控制器，不能被屏蔽
    - 故障
    - 陷阱

外设可屏蔽中断

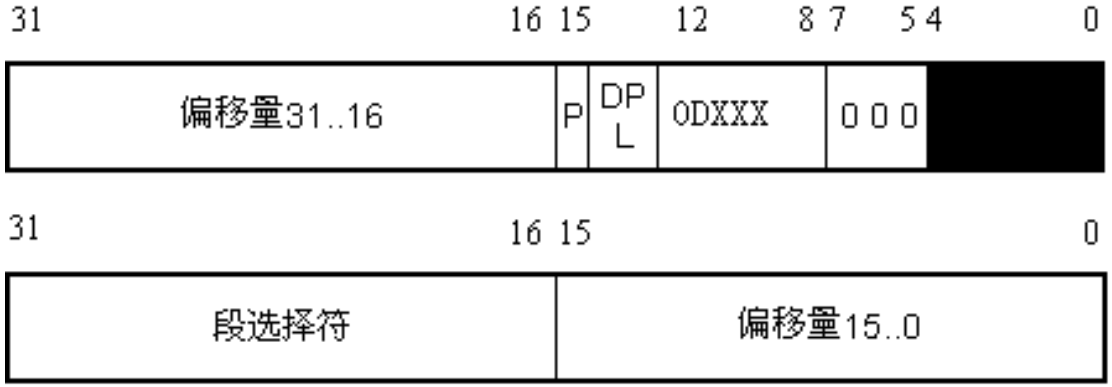
- Intel x86通过两片中断控制器8259A来响应15个外中断源，每个8259A可管理8个中断源。
- 外部设备拥有相应权限时，可以向特定的中断线发送中断请求信号
- 外部I/O请求的屏蔽：
  - 从CPU的角度，清除eflag的中断标志位
  - 从中断控制器的角度，将中断屏蔽寄存器的相应位置位

异常及非屏蔽中断

- 异常就是CPU内部出现的中断,即在CPU执行特定指令时出现的非法情况。
- 非屏蔽中断就是计算机内部硬件出错时引起的异常情况
- Intel把非屏蔽中断作为一种异常来处理
- 在CPU执行一个异常处理程序时，就不再为其他异常或可屏蔽中断请求服务

中断描述符表

- 中断描述符表（IDT）：即中断向量表，每个中断占据一个表项



DPL	段描述符的特权级
偏移量	入口函数地址的偏移量
P	段是否在内存中的标志
段选择符	入口函数所处代码段的选择符
D	标志位，1=32位，0=16位
XXX	3位门类型码

相关汇编指令

- 调用过程指令CALL：
  - **CALL 过程名**
- 调用中断过程的指令INT
  - **INT 中断向量**
- 中断返回指令IRET
  - **IRET**
- 加载中断描述符表的指令LIDT
  - **LIDT 48位的伪描述符**

初始化中断描述符表

- Linux内核在系统的初始化阶段要初始化可编程控制器8259A；将中断描述符表的起始地址装入IDTR寄存器，并初始化表中的每一项
- 当计算机运行在实模式时，中断描述符表被初始化，并由BIOS使用 。
- 真正进入了Linux内核，中断描述符表就被移到内存的另一个区域，并为进入保护模式进行预初始化

< >

IDT表项的设置

- IDT表项的设置通过_set_gaet()函数实现
- 调用该函数在IDT表中插入一个中断门：
  set_intr_gate(unsigned int n, void *addr)
- 调用该函数在IDT表中插入一个陷阱门：
  set_trap_gate(unsigned int n, void *addr)
- 调用该函数在IDT表中插入一个系统门：
  set_system_gate(unsigned int n, void *addr)

初始化陷阱门和系统门

- trap_init()函数用于设置中断描述符表开头的19个陷阱门和系统门
- 这些中断向量都是CPU保留用于异常处理的，例：
- set_trap_gate(0,&divide_error);
  set_trap_gate(1,&debug);
  set_trap_gate(19,&simd_coprocessor_error);
  set_system_gate(SYSCALL_VECTOR,&system_call);

中断门的设置

```
for (i = 0; i< NR_IRQS; i++) {
 int vector = FIRST_EXTERNAL_VECTOR + i;
 if (vector != SYSCALL_VECTOR)
        set_intr_gate(vector, interrupt[i]); }
```

- 中断门的设置是由init_IRQ( )函数中的一段代码完成的：
- 设置时必须跳过用于系统调用的向量0x80
- 中断处理程序的入口地址是一个数组interrupt[]，数组中的每个元素是指向中断处理函数的指针。

中断处理

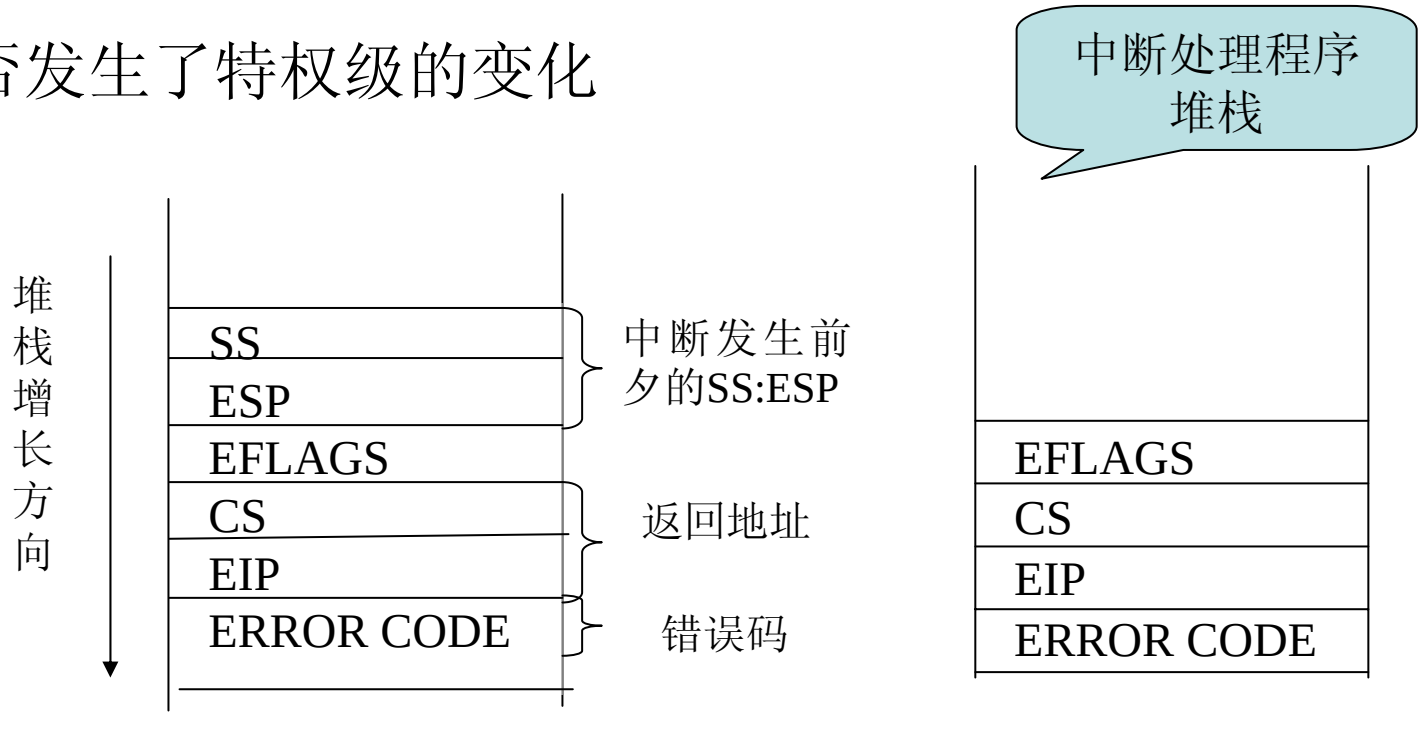
- 中断和异常的硬件处理：
  - 从硬件的角度看CPU如何处理中断和异常
- 中断请求队列的建立：
  - 方便外设共享中断线
- 中断处理程序的执行
- 从中断返回：
  - 调用恢复中断现场的宏RESTORE_ALL，彻底从中断返回

中断和异常的硬件处理

- 当CPU执行了当前指令之后，CS和EIP这对寄存器中所包含的内容就是下一条将要执行指令的虚地址。
- 在对下一条指令执行前，CPU先要判断在执行当前指令的过程中是否发生了中断或异常。
- 如果发生了一个中断或异常，那么CPU将做以下事情：

中断和异常处理中CPU的工作

- 确定所发生中断或异常的向量i（在0～255之间）
- 通过IDTR寄存器找到IDT表，读取IDT表第i项（或叫第i个门）
- 分“段”级、“门”级两步进行有效性检查
- 检查是否发生了特权级的变化



中断请求队列的建立

- 由于硬件条件的限制，很多硬件设备共享一条中断线
- 为方便处理，Linux为每条中断线设置了一个中断请求队列
- 中断服务例程与中断处理程序
- 中断线共享的数据结构
- 注册中断服务例程

< >

中断服务例程与中断处理程序

- 中断服务例程（Interrupt Service Routine）：每个中断请求都有自己单独的中断服务例程
- 中断处理程序：共享同一条中断线的所有中断请求有一个总的中断处理程序
- 在Linux中，15条中断线对应15个中断处理程序

中断线共享的数据结构

```
struct irqaction {
  void  (*handler)(int,  void
    *, struct pt_regs *);
  unsigned long flags;
  unsigned long mask;
  const char *name;
  void *dev_id;
  struct irqaction *next;
};
```

- **Handler**：指向一个具体I/O设备的中断服务例程
- **Flags**：用一组标志描述中断线与I/O设备之间的关系。
- **SA_INTERRUPT**：中断处理程序执行时必须禁止中断
- **SA_SHIRQ**：允许其它设备共享这条中断线。

中断线共享的数据结构

```
struct irqaction {
  void  (*handler)(int,  void
    *, struct pt_regs *);
  unsigned long flags;
  unsigned long mask;
  const char *name;
  void *dev_id;
  struct irqaction *next;
};
```

- SA_SAMPLE_RANDOM：内核可以用它做随机数产生器。
- SA_PROBE：内核正在使用这条中断线进行硬件设备探测。
- Name：I/O设备名
- dev_id：指定I/O设备的主设备号和次设备号(参见第9章)。
- Next：指向irqaction描述符链表的下一个元素

注册中断服务例程

•int request_irq(unsigned int irq, void (*handler)(int, void *, struct pt_regs *),
unsigned long irqflags,
const char * devname,
void *dev_id)

- 初始化IDT表之后，必须通过 request_irq() 函数将相应的中断服务例程挂入中断请求队列，即对其进行注册
- 在关闭设备时，必须通过调用free_irq()函数释放所申请的中断请求号

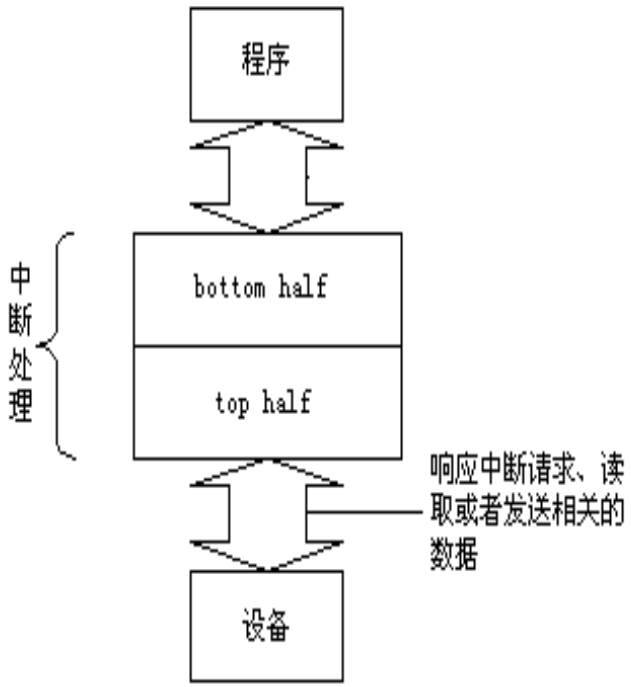
中断处理程序的执行

- CPU从中断控制器的一个端口取得中断向量I
- 根据I从中断描述符表IDT中找到相应的中断门
- 从中断门获得中断处理程序的入口地址
- 判断是否要进行堆栈切换
- 调用do_IRQ()对所接收的中断进行应答，并禁止这条中断线
- 调用handle_IRQ_event（）来运行对应的中断服务例程

从中断返回

- 当处理所有外设中断请求的函数do_IRQ()执行时，内核栈顶包含的就是do_IRQ()的返回地址，这个地址指向ret_from_intr
- 从中断返回时，CPU要调用恢复中断现场的宏RESTORE_ALL，彻底从中断返回。

中断的下半部处理机制



- 中断服务例程在中断请求关闭的条件下执行,避免嵌套使中断控制复杂化
- 系统不能长时间关中断运行，因此内核应尽可能快的处理完中断请求，尽其所能把更多的处理向后推迟
- 内核把中断处理分为两部分：上半部（top half）和下半部（bottom half），上半部内核立即执行，而下半部留着稍后处理

小任务机制

```
struct tasklet_struct {
Struct tasklet_struct *next;
unsigned long state;
atomic_t count;
void (*func) (unsigned long);
unsigned long data;
};
```

- 小任务是指待处理的下半部，其数据结构为tasklet_struct，每个结构代表一个独立的小任务
- 小任务既可以静态地创建，也可以动态地创建

< >

编写并调度自己的小任务

- void tasklet_handler(unsigned long data)
- 小任务不能睡眠，不能在小任务中使用信号量或者其它产生阻塞的函数。但它运行时可以响应中断
- 通过调用tasklet_schedule()函数并传递给它相应的tasklet_struct指针，该小任务就会被调度以便适当的时候执行：

  tasklet_schedule(&my_tasklet)
- 在小任务被调度以后，只要有机会它就会尽可能早的运行

下半部

- 下半部是一个不能与其他下半部并发执行的高优先级小任务
- bh_base是一个指向下半部的指针数组，用于组织所有下半部
- bh_base数组共有32项，每一项都是一种下半部

Linux常用的下半部

下半部	外部设备
TIMER_BH	定时器
TQUEUE_BH	周期性任务队列
SERIAL_BH	串行端口
IMMEDIATE_BH	立即任务队列

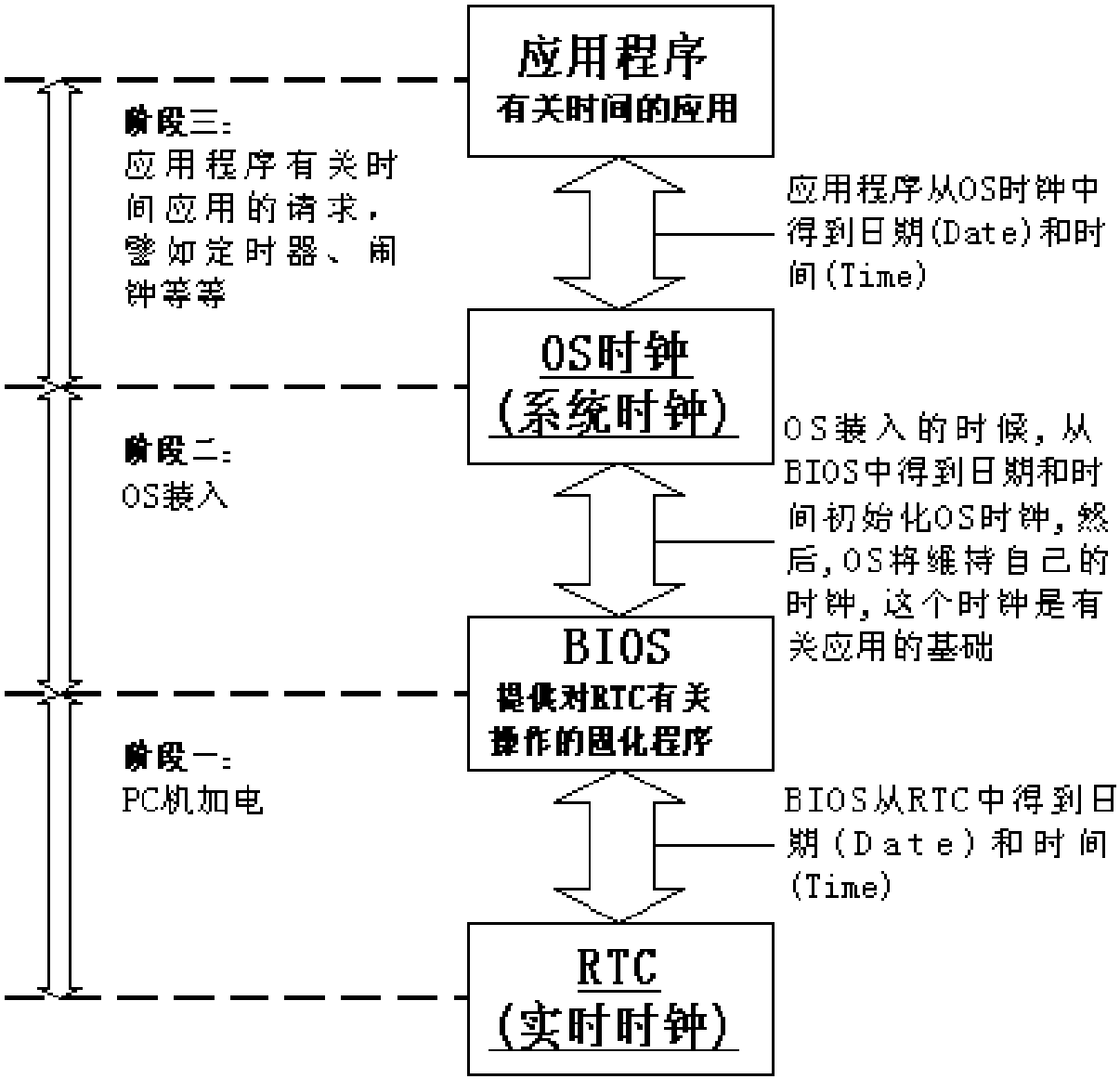
任务队列

- 任务队列就是指以双向队列形式连接起来的任务链表，每一个链表元素都描述了一个可执行的内核任务
- 三个特殊的任务队列：
  - tq _immediate任务队列，由IMMEDIATE_BH下半部运行，该队列中包括要执行的内核函数和标准的下半部。
  - tq_timer任务队列，由TQUEUE_BH下半部运行，每次时钟中断都激活这个下半部。
  - tq_disk任务队列，用于块设备任务。

中断的应用－时钟中断

- 大部分PC机中有两个时钟源，分别是实时时钟（RTC）和 操作系统（OS）时钟
- 实时时钟也叫硬件时钟，它靠电池供电，即使系统断电，也可以维持日期和时间。
- RTC和OS时钟之间的关系通常也被称作操作系统的时钟运作机制
- 不同的操作系统，其时钟运作机制也不同

时钟运作机制


应用程序
有关时间的应用
阶段三：
应用程序有关时间应用的请求，譬如定时器、闹钟等等
应用程序从OS时钟中得到日期(Date)和时间(Time)
OS时钟
(系统时钟)
阶段二：
OS装入
OS装入的时候，从BIOS中得到日期和时间初始化OS时钟，然后，OS将维持自己的时钟，这个时钟是有关应用的基础
BIOS
提供对RTC有关操作的固化程序
阶段一：
PC机加电
BIOS从RTC中得到日期(Date)和时间(Time)
RTC
(实时时钟)

Linux时间系统

- OS时钟是由可编程定时/计数器产生的输出脉冲触发中断而产生的
- 操作系统的“时间基准”由设计者决定，Linux的时间基准是1970年1月1日凌晨0点
- OS时钟记录的时间就是系统时间。系统时间以“时钟节拍”为单位
- Linux中用全局变量jiffies表示系统自启动以来的时钟节拍数目

时钟中断处理程序

- 每一次时钟中断的产生都触发下列几个主要的操作：
  - 自系统启动以来所花费的时间
  - 更新时间和日期
  - 确定当前进程在CPU 上已运行了多长时间，如果已经超过了分配给它的时间，则抢占它
  - 更新资源使用统计数
  - 检查定时器时间间隔是否已到，如果是，则调用适当的函数

时钟中断的下半部处理

- timer_bh( )函数与TIMER_BH 下半部相关联，它在每个时钟节拍都被激活
- TIMER_BH 下半部以关中断调用update_times( )函数,该函数会以关中断来更新xtime
- 更新了系统时钟xtime之后，update_times( )再次打开中断

定时器及应用

```
struct timer_list {
    struct list_head  entry;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long);
};
```

- 定时器是管理内核所花时间的基础，也被称为动态定时器或内核定时器
- 定时器的使用：执行一些初始化工作，设置一个到期时间，指定到时后执行的函数，然后激活定时器就可以了
- 定时器由timer_list结构表示

定时器的使用

- 定义定时器：
  - struct timer_list my_timer;
- 初始化定时器：
  - init_timer(&my_timer);
- 激活定时器：
  - add_timer(&my_timer);
- 如果需要在定时器到期前停止定时器，可以使用del_timer()函数：
  - del_timer(&my_timer);

定时器的执行与应用

- 内核在时钟中断发生后执行定时器，定时器在下半部中被run_timer_list( )执行
- 定时器的使用：
  - 例：创建和使用进程延时

```
timeout = 2 * HZ; /*1HZ等于100，因此为2000ms*/
set_current_state(TASK_INTERRUPTIBLE);
remaining = schedule_timeout(timeout);
```

内核用定时器实现进程的延时，调用schedule_timeout( )函数，该函数执行下列语句：struct timer_list timer;

```
expire = timeout + jiffies;
init_timer(&timer);
timer.expires = expire;
timer.data = (unsigned long) current;
timer.function = process_timeout;
add_timer(&timer);
schedule( );      /* 进程被挂起直到定时器到期 */
del_timer_sync(&timer);
timer.timeout = expire - jiffies;
return (timeout < 0 ? 0 : timeout);
```

当延时到期时，内核执行下列函数：

```
void process_timeout(unsigned long data)
{  struct task_struct * p = (struct task_struct *) data;
        wake_up_process(p);
}
```

“内核之旅”网站

- http://www.kerneltravel.net/
- 电子杂志栏目是关于内核研究和学习的资料
- 第八期“中断”，将向读者依次解释中断概念，解析Linux中的中断实现机理以及Linux下中断如何被使用。

第六章 系统调用

- 系统调用与API、系统命令、内核函数
- 系统调用处理程序及服务例程
- 封装例程
- 添加新的系统调用
- 系统调用实例

系统调用—内核的出口

- 系统调用，顾名思义，说的是操作系统提供给用户程序调用的一组“特殊”接口。
- 从逻辑上来说，系统调用可被看成是一个内核与用户空间程序交互的接口——它好比一个中间人，把用户进程的请求传达给内核，待内核把请求处理完毕后再将处理结果送回给用户空间。

系统调用与API

- Linux的应用编程接口（API）遵循 POSIX标准
- 应用编程接口(API)其实是一组函数定义，这些函数说明了如何获得一个给定的服务；而系统调用是通过软中断向内核发出一个明确的请求
- API有可能和系统调用的调用形式一致
- API和系统调用关注的都是函数名、参数类型及返回代码的含义
- 系统调用的实现是在内核完成的，而用户态的函数是在函数库中实现的

系统调用与系统命令

- 系统命令相对应用编程接口更高一层，每个系统命令都是一个可执行程序，比如ls、hostname等，
- 系统命令的实现调用了系统调用
- 通过strace ls或strace hostname命令可以查看系统命令所调用的系统调用

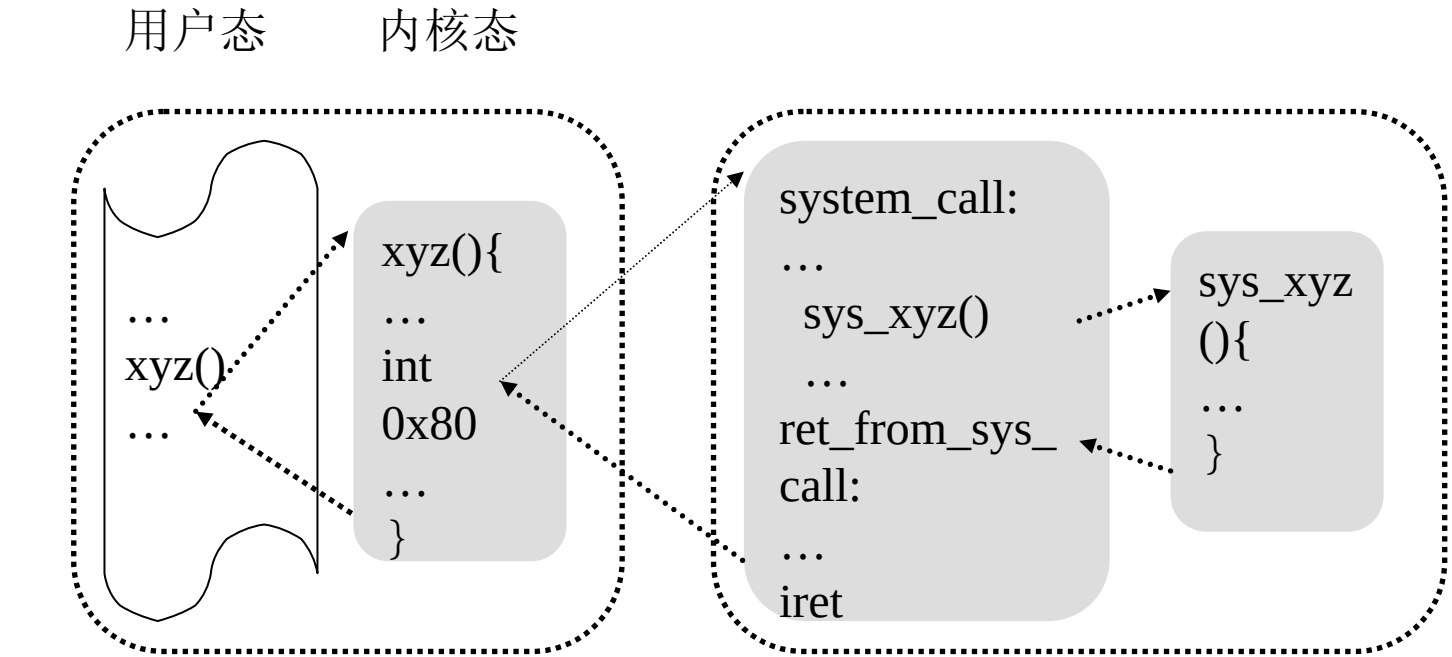
系统调用与内核函数

- 内核函数在形式上与普通函数一样，但它是在内核实现的，需要满足一些内核编程的要求
- 系统调用是用户进程进入内核的接口层，它本身并非内核函数，但它是由内核函数实现的
- 进入内核后，不同的系统调用会找到各自对应的内核函数，这些内核函数被称为系统调用的**“服务例程”**

系统调用处理程序及服务例程

- 当用户态的进程调用一个系统调用时，CPU切换到内核态并开始执行一个内核函数
- 系统调用处理程序执行下列操作：
  - 在内核栈保存大多数寄存器的内容
  - 调用所谓系统调用服务例程的相应的C函数来处理系统调用
  - 通过ret_from_sys_call( )函数从系统调用返回

调用一个系统调用


用户态
内核态
…
xyz()
…
xyz(){
…
int
0x80
…
}
system_call:
…
sys_xyz()
…
ret_from_sys_
call:
…
iret
sys_xyz
(){
…
}
在应用程序
调用中的
系统调用
在libc标准库
中的封装例程
系统调用
处理程序
系统调用
服务例程

初始化系统调用

- 内核初始化期间调用trap_init( ) 函数建立IDT表中128号向量对应的表项：
  - set_system_gate(0x80， &system_call);
- 该调用把下列值装入该门描述符的相应域 ：
  - 段选择子：
  - 偏移量：指向system_call( )异常处理程序
  - 类型：置为15，表示该异常是一个陷阱
  - DPL（描述符特权级）：置为3，这就允许用户态进程调用这个异常处理程序

system_call( )函数

❖ **system_call( )函数实现了系统调用处理程序：**

- 它首先把系统调用号和该异常处理程序用到的所有CPU寄存器保存到相应的栈中
- 把当前进程PCB的地址存放在ebx中
- 对用户态进程传递来的系统调用号进行有效性检查。若调用号大于或等于NR_syscalls，系统调用处理程序终止
- 若系统调用号无效，函数就把-ENOSYS值存放在栈中eax寄存器所在的单元，再跳到ret_from_sys_call( )
- 根据eax中所包含的系统调用号调用对应的特定服务例程

参数传递

- 每个系统调用至少有一个参数，即通过eax寄存器传递来的系统调用号
- 用寄存器传递参数必须满足两个条件：
  - 每个参数的长度不能超过寄存器的长度
  - 参数的个数不能超过6个（包括eax中传递的系统调用号），否则，用一个单独的寄存器指向进程地址空间中这些参数值所在的一个内存区即可
- 在少数情况下，系统调用不使用任何参数
- 服务例程的返回值必须写到eax寄存器中

跟踪系统调用的执行

- 分析系统调用有两种方法：
  - 查看entry.s中的代码细节，阅读相关的源码来分析其运行过程；
  - 借助一些内核调试工具，动态跟踪执行路径。
- 结合用户空间的执行路径，该程序的执行大致可归结为以下几个步骤：

1、程序调用libc库的封装函数
2、调用软中断 int 0x80 进入内核。
3、在内核中首先执行system_call函数，接着根据系统调用号在系统调用表中查找到对应的系统调用服务例程
4、执行该服务例程
5、执行完毕后，转入ret_from_sys_call例程，从系统调用返回

封装例程

- **libc库中如何对不同的服务例程进行封装？**
- Linux定义了从_syscall0到_syscall5的六个宏
- 每个宏名字的数字0到5对应着系统调用所用的参数个数(系统调用号除外)
- 每个宏严格地需要2+2×n个参数，n是系统调用的参数个数。另外两个参数指明系统调用的返回值类型和名字；每一对参数指明相应的系统调用参数的类型和名字
- 例：fork( )的封装例程：_syscall0(int，fork)
- 在用户态进行系统调用时，要进行用户态堆栈到内核态堆栈的切换，但在内核中进行系统调用时，不用进行堆栈切换

添加新的系统调用

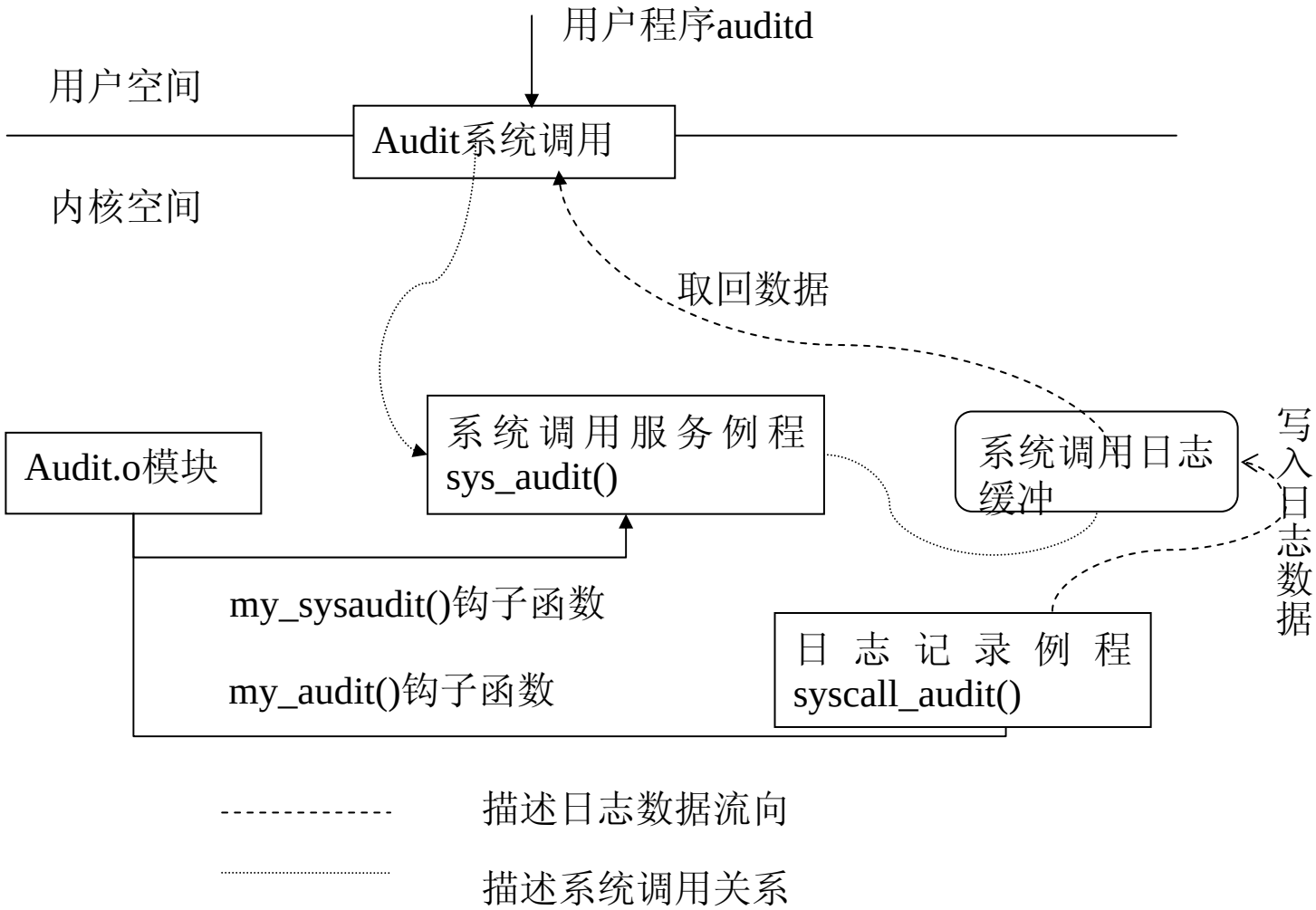
- 添加系统调用的步骤：
  - 添加系统调用号。系统调用号在unistd.h文件中定义，以“__NR_”开头
  - 在系统调用表（在entry.S中）中添加自己的服务例程sys_mysyscall
  - 实现系统调用服务例程：把sys_mysyscall加在kernel目录下的系统调用文件sys.c中
  - 重新编译内核
  - 编写用户态程序

系统调用实例

- 利用系统调用实现一个调用日志收集系统，实时获取系统调用日志
- 本实例需要完成以下几个基本功能：
  - 记录系统调用日志，将其写入缓冲区（内核中），以便用户读取（由syscall_audit()实现）；
  - 建立新的系统调用，将内核缓冲中的系统调用日志返回到用户空间（由Sys_audit()实现）
  - 循环利用系统调用，以便能动态实时返回系统调用的日志（由 auditd() 实现）

代码结构体系介绍

- 日志记录例程Syscall_audit()：
  - 内核态的服务例程，负责记录系统调用的运行日志，包括调用时刻、调用者PID、程序名等
- 服务例程Sys_audit()：
  - 新添加的系统调用，其功能是从缓冲区中取数据返回用户空间
- 用户空间服务程序auditd：
  - 取回系统中搜集到的系统调用日志信息


用户程序auditd
用户空间
Audit系统调用
内核空间
取回数据
系统调用服务例程
sys_audit()
系统调用日志
缓冲
写入日志数据
Audit.o模块
my_sysaudit()钩子函数
my_audit()钩子函数
日志记录例程
syscall_audit()
描述日志数据流向
描述系统调用关系

程序的体系结构图

把代码集成到内核中

- 改entry.S汇编代码，在系统调用表中加入新的系统调用，然后在系统调用入口中加入跳转到日志记录服务例程
- 填加代码文件audit.c，该文件中包含syscall_audit与系统调用sys_audit两个函数体
- 修改i386_ksyms.c文件，在最后加入

```
EXPORT_SYMBOL(my_audit);
  extern int(*my_sysaudit)(unsigned char,unsigned char*,unsigned short,unsigned char);
```

- `EXPORT_SYMBOL(my_sysaudit);`

添加系统调用Step By Step

- 修改/arch/i386/kernel/entry.S 文件
- 添加audit.c文件到/arch/i386/kernel/目录下
- 修改/arch/i386/kernel/i386-kysms.c文件，在其中导出my_audit与my_sysaudit两个钩子函数
- 修改/arch/i386/kernel/Makefile文件，将audit.c编译入内核

重新编译内核 Step By Step

- 编写名为audit的模块
- 编译并加载模块 insmod audit.o
- 编写一个用户程序deamon循环调用audit系统调用，并把搜集到的信息打印到屏幕上。

“内核之旅”网站

- http://www.kerneltravel.net/
- 电子杂志栏目是关于内核研究和学习的资料
- 第四期“系统调用”，本期重点和大家讨论系统调用机制。其中涉及到了一些及系统调用的性能、上下文深层问题，同时也穿插着讲述了一些内核调试方法。
- 实验部分我们利用系统调用与相关内核服务完成了一个搜集系统调用序列的特定任务
- 下载其代码进行调试

第七章 内核中的同步

- 临界区和竞争状态
- 内核同步措施
- 并发控制

临界区和竞争状态

- 什么是临界区（*critical regions*）？
  - 就是访问和操作共享数据的代码段，这段代码必须被**原子地**执行
- 什么是竞争状态?
  - 多个内核任务同时访问同一临界区
- 什么是同步?
  - 避免并发和防止竞争状态称为*同步*（*synchronization*）

临界区举例

- 考虑一个非常简单的共享资源的例子：一个全局整型变量和一个简单的临界区，其中的操作仅仅是将整型变量的值增加1：

  i++

- 该操作可以转化成下面三条机器指令序列：
  - (1) 得到当前变量i的值并拷贝到一个寄存器中
  - (2)将寄存器中的值加1
  - (3) 把i的新值写回到内存中

临界区举例

可能的实际执行结果：

内核任务1	内核任务2
获得 i(1)	---
---	获得 i(1)
增加 i(1->2)	---
---	增加 i(1->2)
写回 i(2)	---
---	写回 i(2)

期望的结果

内核任务1	内核任务2
获得i(1)	---
增加 i(1->2)	---
写回 i(2)	---
	获得 i(2)
	增加 i(2->3)
	写回 i(3)

共享队列和加锁

- 当共享资源是一个复杂的数据结构时，竞争状态往往会使该数据结构遭到破坏。
- 对于这种情况，锁机制可以避免竞争状态正如门锁和门一样，门后的房间可想象成一个临界区。
- 在一个指定时间内，房间里只能有个一个内核任务存在，当一个任务进入房间后，它会锁住身后的房门；当它结束对共享数据的操作后，就会走出房间，打开门锁。如果另一个任务在房门上锁时来了,那么它就必须等待房间内的任务出来并打开门锁后，才能进入房间。

共享队列和加锁

- 任何要访问队列的代码首先都需要占住相应的锁，这样该锁就能阻止来自其它内核任务的并发访问：

任务 1

试图锁定队列

成功：获得锁

访问队列…

为队列解除锁

…

任务2

试图锁定队列

失败：等待…

等待…

等待…

成功：获得锁

访问队列…

为队列解除锁

确定保护对象

- 找出哪些数据需要保护是关键所在
- 内核任务的局部数据仅仅被它本身访问，显然不需要保护
- 如果数据只会被特定的进程访问，也不需加锁
- 大多数内核数据结构都需要加锁：若有其它内核任务可以访问这些数据，那么就给这些数据加上某种形式的锁；若任何其它东西能看到它，那么就要锁住它

死　锁

- 死锁产生的条件：有一个或多个并发执行的内核任务和一个或多个资源，每个任务都在等待其中的一个资源，但所有的资源都已经被占用。所有任务都在相互等待，但它们永远不会释放已经占有的资源，于是任何任务都无法继续
- 典型的死锁：
  - 四路交通堵塞
  - 自死锁：一个执行任务试图去获得一个自己已经持有的锁

死锁的避免

- 加锁的顺序是关键。使用嵌套的锁时必须保证以相同的顺序获取锁，这样可以阻止致命拥抱类型的死锁
- 防止发生饥饿
- 不要重复请求同一个锁。
- 越复杂的加锁方案越有可能造成死锁，因此设计应力求简单

并发执行的原因

- 中断——中断几乎可以在任何时刻异步发生，也可能随时打断正在执行的代码。
- 内核抢占——若内核具有抢占性，内核中的任务就可能会被另一任务抢占
- 睡眠及与用户空间的同步——在内核执行的进程可能会睡眠，这将唤醒调度程序，导致调度一个新的用户进程执行
- 对称多处理——两个或多个处理器可以同时执行代码

内核同步措施

- 为了避免并发，防止竞争。内核提供了一组同步方法来提供对共享数据的保护
- 原子操作
- 自旋锁
- 信号量

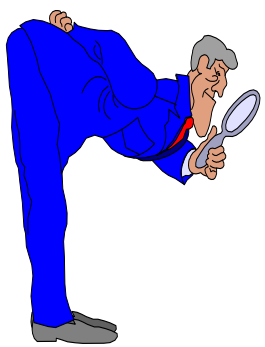
原子操作

- 原子操作可以保证指令以*原子的方式*被执行
- 两个原子操作绝对不可能并发地访问同一个变量
- Linux内核提供了一个专门的atomic_t类型（一个24位原子访问计数器）和一些专门的函数，这些函数作用于atomic_t类型的变量

自旋锁

- 自旋锁是专为防止多处理器并发而引入的一种锁，它在内核中大量应用于中断处理等部分，而对于单处理器来说，可简单采用关闭中断的方式防止中断处理程序的并发执行
- 自旋锁最多只能被一个内核任务持有，若一个内核任务试图请求一个已被持有的自旋锁，那么这个任务就会一直进行忙循环，也就是旋转，等待锁重新可用

自旋锁



- 设计自旋锁的初衷是在短期间内进行轻量级的锁定。一个被持有的自旋锁使得请求它的任务在等待锁重新可用期间进行自旋，所以自旋锁不应该被持有时间过长
- 自旋锁在内核中主要用来防止多处理器中并发访问临界区，防止内核抢占造成的竞争
- 自旋锁不允许任务睡眠，持有自旋锁的任务睡眠会造成自死锁，因此自旋锁能够在中断上下文中使用

信号量

- Linux中的信号量是一种睡眠锁。若有一个任务试图获得一个已被持有的信号量时，信号量会将其推入等待队列，然后让其睡眠。这时处理器获得自由而去执行其它代码。当持有信号量的进程将信号量释放后，在等待队列中的一个任务将被唤醒，从而便可以获得这个信号量
- 信号量具有睡眠特性，适用于锁会被长时间持有的情况，只能在进程上下文中使用

信号量

```
struct semaphore {
atomic_t count;
int sleepers;
wait_queue_head_t wait;
}
```

信号量的定义

信号量的使用

```
static DECLARE_MUTEX(mr_sem);
/ *声明并初始化互斥信号量*/
if(down_interruptible(&mr_sem))
/* 信号被接收 , 信号量还未获取 */

/*临界区…*/
up(&mr_sem);
```

信号量与自旋锁的比较

需求	建议的加锁方法
低开销加锁	优先使用自旋锁
短期锁定	优先使用自旋锁
长期加锁	优先使用信号量
中断上下文中加锁	使用自旋锁
持有锁时需要睡眠、调度	使用信号量

并发控制实例

- 假设存在这样一个的内核共享资源—链表。另外我们构造一个内核多任务访问链表的场景：内核线程向链表加入新节点；内核定时器定时删除接点；系统调用销毁链表。
- 上面三种内核任务并发执行时，有可能会破坏链表数据的完整性，所以我们必须对链表进行同步访问保护，以确保数据一致性。

内核任务及其之间的并发关系

- **系统调用：**是用户程序通过门机制来进入内核执行的内核例程，它运行在内核态，处于进程上下文中，可以认为是代表用户进程的内核任务
- **内核线程：**内核线程可以理解成在内核中运行的特殊进程，它有自己的“进程上下”，
- **定时器任务队列：**任务队列属于下半部，主要有调度队列、定时器队列和及时队列等三种任务队列

内核任务及其之间的并发关系

- 系统调用和内核线程可能和各种内核任务并发执行，除了中断（定时器任务队列属于软中断范畴）抢占它产生并发外，它们还有可能自发性地主动睡眠（比如在一些阻塞性的操作中），于是放弃处理器，从而重新调度其它任务，所以系统调用和内核线程除与定时器任务队列发生竞争，也会与其他（包括自己）系统调用与内核线程发生竞争。

实现机制

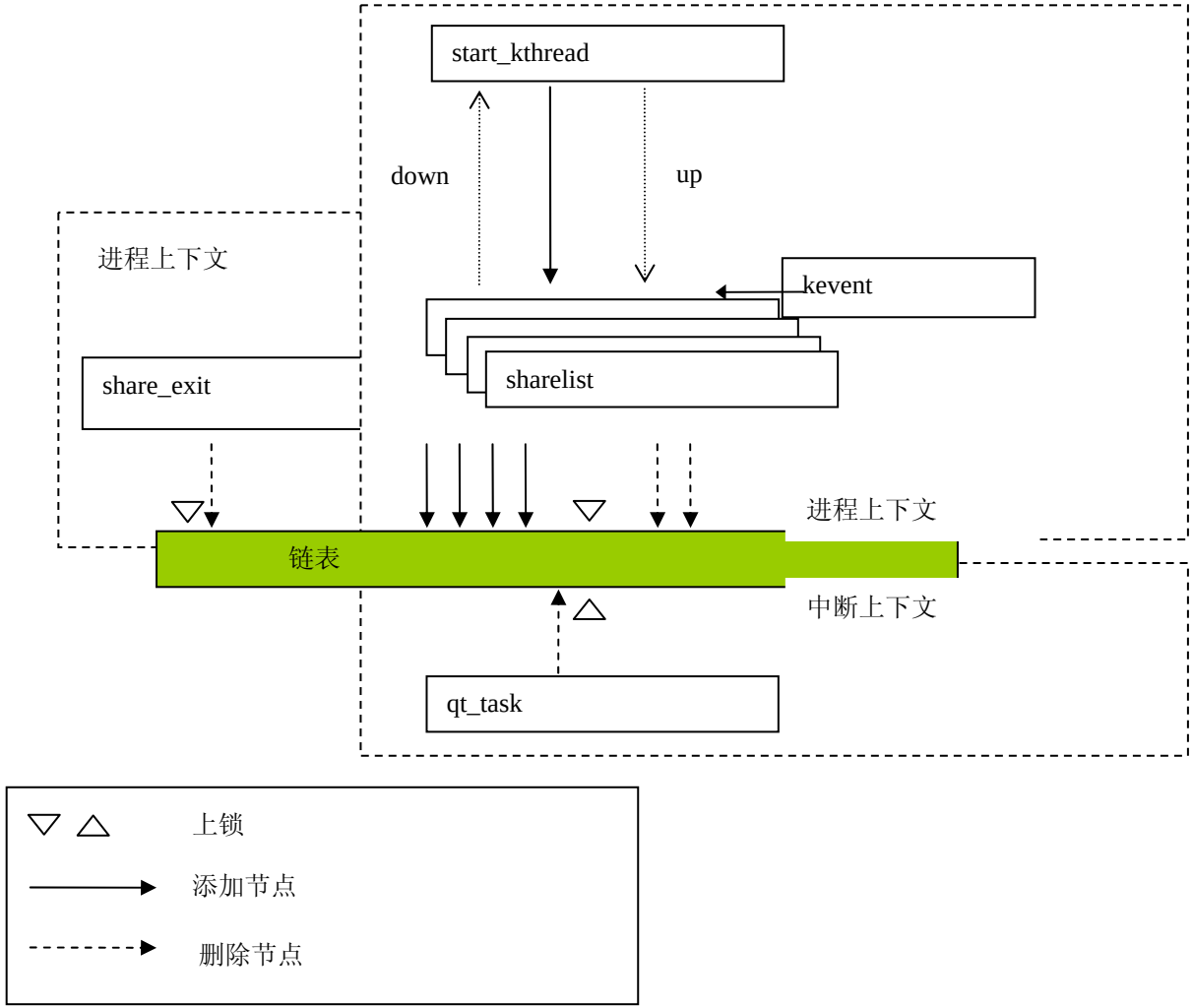
- 主要的共享资源是链表（mine）,操作它的内核任务有三个：一是100个内核线程（sharelist），它们负责从表头将新节点（struct my_struct）插入链表。二是定时器任务(qt_task)，它负责每个时钟节拍时从链表头删除一个节点。三是系统调用share_exit，它负责销毁链表并卸载模块。

实现机制

- 内核线程sharelist：
  - 该函数是作为内核线程由keventd调度执行的，作用是向链表中加入新节点
- start_kthread：
  - 该函数用来构建内核线程Sharelist的封装函数kthread_launcher，并启动它
- kthread_launcher：
  - 该函数作用仅仅是通过kernel_thread方法启动内核线程sharelist

实现机制

- **qt_task：**
  - 该函数删除链表节点，作为定时器任务运行
- **share_init：**
  - 该函数是我们的模块注册函数，也是通过它启动定时器任务和内核线程
- **share_exit：**
  - 这是模块注销函数，负责销毁链表


start_kthread
down
up
进程上下文
kevent
share_exit
sharelist
进程上下文
链表
中断上下文
qt_task
上锁
添加节点
删除节点

关键代码解释

- 链表是内核开发中的常见数据组织形式，为了方便开发和统一结构，内核提供了一套接口来操作链表，我们用到的接口其主要功能为：
  - LIST_HEAD：声明链表结构
  - list_add()：添加节点到链表
  - list_del()：删除节点
  - list_entry()：遍历链表
- 任务队列结构为 struct tq_struct
- kill_proc，该函数在模块注销时被调用，其主要作用有两个：第一杀死我们生成的内核线程；第二告诉keventd回收相关子线程，以免产生“残疾”子线程序

一步一步

- 对该模块的实际操作步骤如下：
  - Make 编译模块
  - Insmod sharelist.o 加载模块
  - Rmmod sharelist 卸载模块
  - Dmesg 观察结果

“内核之旅”网站

- http://www.kerneltravel.net/
- 第七期“内核中的调度与同步”为大家介绍内核中存在的各种任务调度机理以及它们之间的逻辑关系（这里将覆盖进程调度、推后执行、中断等概念、内核线程），在此基础上向大家解释内核中需要同步保护的根本原因和保护方法
- 提供了一个内核共享链表同步访问的例子，帮助大家理解内核编程中的同步问题。
- 下载其代码进行调试

第八章 文件系统

- Linux文件系统
- 虚拟文件系统
- 文件系统的注册、安装与卸载
- 页缓冲区
- 文件的打开与读写
- 文件系统的编写

Linux文件系统基础

- **Linux的文件结构**
  - 简单介绍Linux下文件存放在存储设备上的组织方法
- **Linux的文件系统**
  - 文件所在的物理空间
- **Linux下的文件类型**
- **访问权限和文件模式**

Linux的文件结构

- 文件结构是文件存放在磁盘等存贮设备上的组织方法。主要体现在对文件和目录的组织上。
- Linux使用标准的目录结构－树型结构，无论操作系统管理几个磁盘分区，这样的目录树只有一个
- 制定这样一个固定的目录规划有助于对系统文件和不同的用户文件进行统一管理

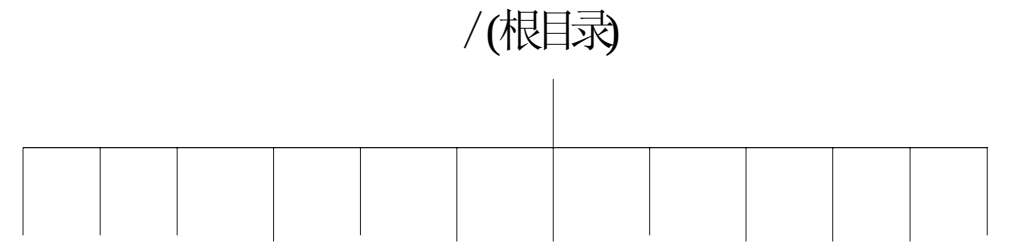


图8.1 Linux目录树结构

Linux文件系统

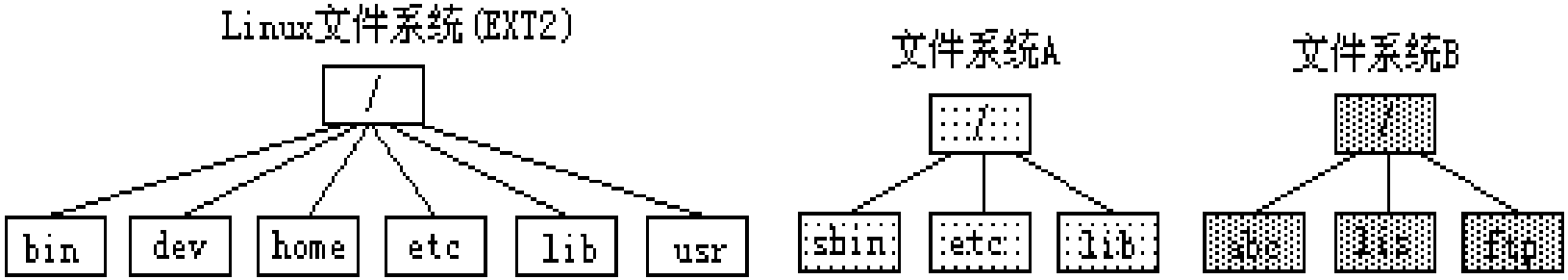
- 文件系统：文件存在的物理空间，Linux系统中每个分区都是一个文件系统，都有自己的目录层次结构
- Linux文件系统使用索引节点来记录文件信息，系统给每个索引节点分配了一个号码，称为**索引节点号**。文件系统正是靠这个索引节点号来识别一个文件

软链接和硬链接

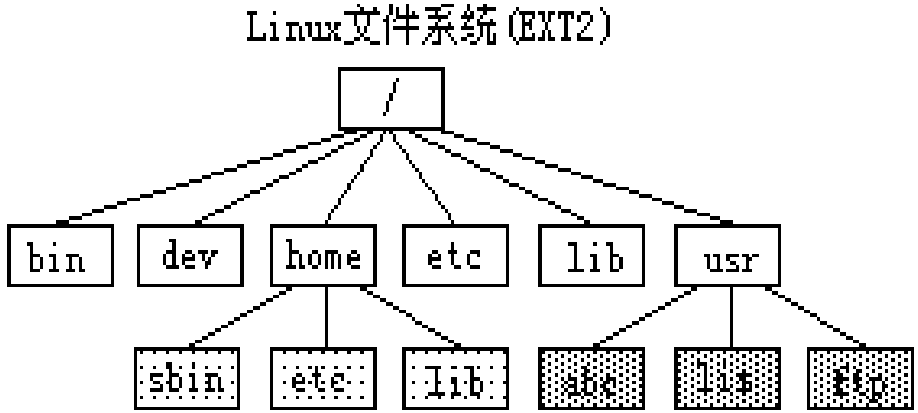
- 可以用链接命令ln（Link）对一个已存在的文件再建立一个新的链接，而不复制文件的内容
- 硬链接(hard link)：让一个文件对应一个或多个文件名，或者说把我们使用的文件名和文件系统使用的节点号链接起来，这些文件名可以在同一目录或不同目录
- 软链接（也叫符号链接）：是一种特殊的文件，这种文件包含了另一个文件的任意一个路径名。这个路径名指向位于任意一个文件系统的任意文件，甚至可以指向一个不存在的文件

安装文件系统

- 将一个文件系统的顶层目录挂到另一个文件系统的子目录上，使它们成为一个整体，称为“安装（mount）”。把该子目录称为“安装点(mount point)”
- EXT2是Linux的标准文件系统，系统把它的磁盘分区做为系统的根文件系统，EXT2以外的文件系统则安装在根文件系统下的某个目录下，成为系统树型结构中的一个分枝
- 安装一个文件系统用mount命令



(a) 安装前的三个独立的文件系统



(b) 安装后的文件系统

文件系统的安装

文件类型

- Linux下的主要文件类型：
  - 常规文件：文本文件和二进制文件
  - 目录文件：将文件的名称和它的索引节点号结合在一起的一张表
  - 设备文件：每种I/O设备对应一个设备文件
  - 管道文件：主要用于在进程间传递数据，又称先进先出(FIFO)文件
  - 链接文件：又称符号链接文件，它提供了共享文件的一种方法

访问权限和文件模式

- Linux给文件设定了一定的访问权限
- Linux对文件的访问设定了三级权限：文件所有者，与文件所有者同组的用户，其他用户。对文件的访问主要是三种处理操作：读取、写入和执行

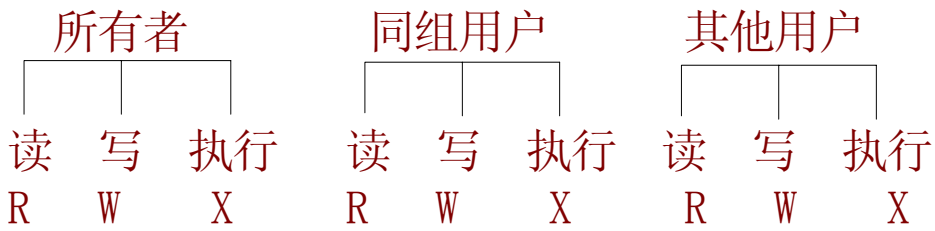


图8.3 文件访问权和访问模式

虚拟文件系统

- 虚拟文件系统的引入
- VFS中的数据结构
- VFS超级块数据结构
- VFS的索引节点
- 目录项对象

虚拟文件系统的引入

- Linux最初采用Minix的文件系统，其大小限于64MB，文件名长度也限于14个字节
- Linux经过一段时间的改进和发展，特别是吸取了Unix文件系统的经验，最后形成了现在的Ext2文件系统
- 为了支持其他各种不同的文件系统，Linux提供了一种统一的框架，就是所谓的虚拟文件系统转换（Virtual Filesystem Switch），简称**虚拟文件系统(VFS)**。

虚拟文件系统的引入

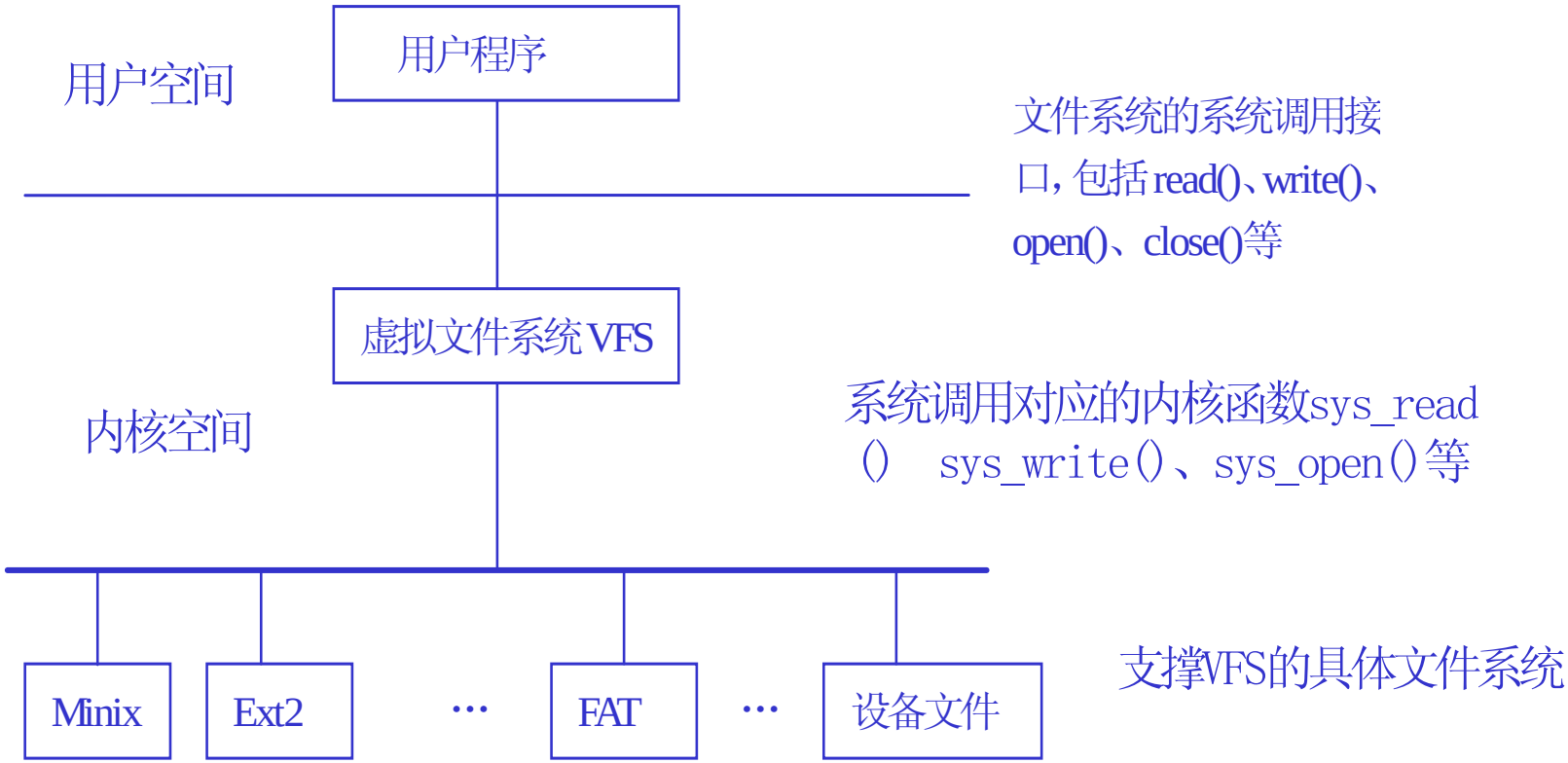


图8.4 VFS与具体文件系统之间的关系

VFS中的数据结构

- **超级块（superblock）对象:** 存放系统中已安装文件系统的有关信息
- **索引节点（inode）对象:** 存放关于具体文件的一般信息
- **目录项（dentry）对象:** 存放目录项与对应文件进行链接的信息
- **文件(file)对象:** 存放打开文件与进程之间进行交互的有关信息

VFS超级块数据结构

- 超级块用来描述整个文件系统的信息。每个具体的文件系统都有各自的超级块
- VFS超级块是各种具体文件系统在安装时建立的，并在卸载时被自动删除，其数据结构是 super block
- 所有超级块对象以双向环形链表的形式链接在一起
- 与超级块关联的方法就是超级块操作表。这些操作是由数据结构super_operations来描述

```
struct super_block
{
   kdev_t s_dev;      /*具体文件系统的块设备标识符*/
   unsigned long s_blocksize; /*以字节为单位数据块的大小*/
   unsigned char s_blocksize_bits;   /*块大小的值占用的位数*/
   …
   struct list_head   s_list;   /*指向超级块链表的指针*/
   struct file_system_type *s_type;
         /*指向文件系统的file_system_type 数据结构的指针 */
   struct super_operations *s_op;
          /*指向具体文件系统的用于超级块操作的函数集合 */
   …
  u;    /*一个共用体，其成员是各种文件系统的
                  fsname_sb_info数据结构 */
}
```

VFS的索引节点

- 文件系统处理文件所需要的所有信息都放在称为索引节点的数据结构inode中。
- 在同一个文件系统中，每个索引节点号都是唯一的
- inode 中有两个设备号，i_dev（常规文件的设备号）和i_rdev（某一设备的设备号）

```
struct inode
{
  struct list_head   i_hash;   /*指向哈希链表的指针*/
  struct list_head   i_list;   /*指向索引节点链表的指针*/
  struct list_head   i_dentry;/*指向目录项链表的指针*/
      ...
  unsigned long   i_ino;    /*索引节点号*/
  kdev_t          i_dev;    /*设备标识号 */
  umode_t                   i_mode;   /*文件的类型与访问权限 */
kdev_t       i_rdev;          /*实际设备标识号*/
uid_t   i_uid;         /* 文件拥有者标识号*/
gid_t   i_gid          /*文件拥有者所在组的标识号*/
          ...
struct inode_operations  *i_op;/*指向对该节点进行操作的一组函数*/
struct super_block   *i_sb;      /*指向该文件系统超级块的指针 */
    atomic_t     i_count;      /*当前使用该节点的进程数。*/
    struct file_operations  *i_fop;  /*指向文件操作的指针 */
     ...
    struct vm_area_struct  *i_op
                /*指向对文件进行映射所使用的虚存区指针 */
    struct page  *i_page           /*指向页结构的指针 */
   unsigned long         i_state;     /*索引节点的状态标志*/
 unsigned int          i_flags;     /*文件系统的安装标志*/
   union { /* 联合体结构，其成员指向具体文件系统的inode结构*/
     struct minix_inode_info    minix_i;
     struct ext2_inode_info     ext2_i;
     ...}
```

目录项对象

- 每个文件除了有一个索引节点inode数据结构外，还有一个目录项dentry数据结构。
- dentry结构代表的是逻辑意义上的文件，描述的是文件逻辑上的属性，目录项对象在磁盘上并没有对应的映像
- inode结构代表的是物理意义上的文件，记录的是物理上的属性，对于一个具体的文件系统，其inode结构在磁盘上就有对应的映像
- 一个索引节点对象可能对应多个目录项对象

```
struct dentry {
  atomic_t d_count;          /* 目录项引用计数器 */
  unsigned int d_flags;      /* 目录项标志 */
  struct inode  * d_inode;    /* 与文件名关联的索引节点 */
  struct dentry * d_parent;        /* 父目录的目录项 */
  struct list_head d_hash;         /* 目录项形成的哈希表 */
  struct list_head d_lru;          /*未使用的 LRU 链表 */
  struct list_head d_child;        /*父目录的子目录项所形成的链表 */
  struct list_head d_subdirs;      /* 该目录项的子目录所形成的链表*/
  struct list_head d_alias;        /* 索引节点别名的链表*/
  int d_mounted;                   /* 目录项的安装点 */
  struct qstr d_name;              /* 目录项名（可快速查找） */
  struct dentry_operations  *d_op;/* 操作目录项的函数*/
  struct super_block * d_sb;     /* 目录项树的根（即文件的超级块）*/
  unsigned long d_vfs_flags;
  void * d_fsdata;                 /* 具体文件系统的数据 */
unsigned char d_iname[DNAME_INLINE_LEN]; /* 短文件名 */
......
};
```

与进程相关的文件结构－文件对象

- 进程是通过**文件描述符**来访问文件的
- Linux中专门用了一个**file文件对象**来保存打开文件的文件位置，这个对象称为**打开的文件描述**（open file description）
- file结构中主要保存了文件位置，此外，还把指向该文件索引节点的指针也放在其中。file结构形成一个双链表，称为系统**打开文件表**。

与进程相关的文件结构－用户打开文件表

- **文件描述符**是用来描述打开的文件的。每个进程用一个files_struct结构来记录文件描述符的使用情况，这个files_struct结构称为**用户打开文件表**，它是进程的私有数据

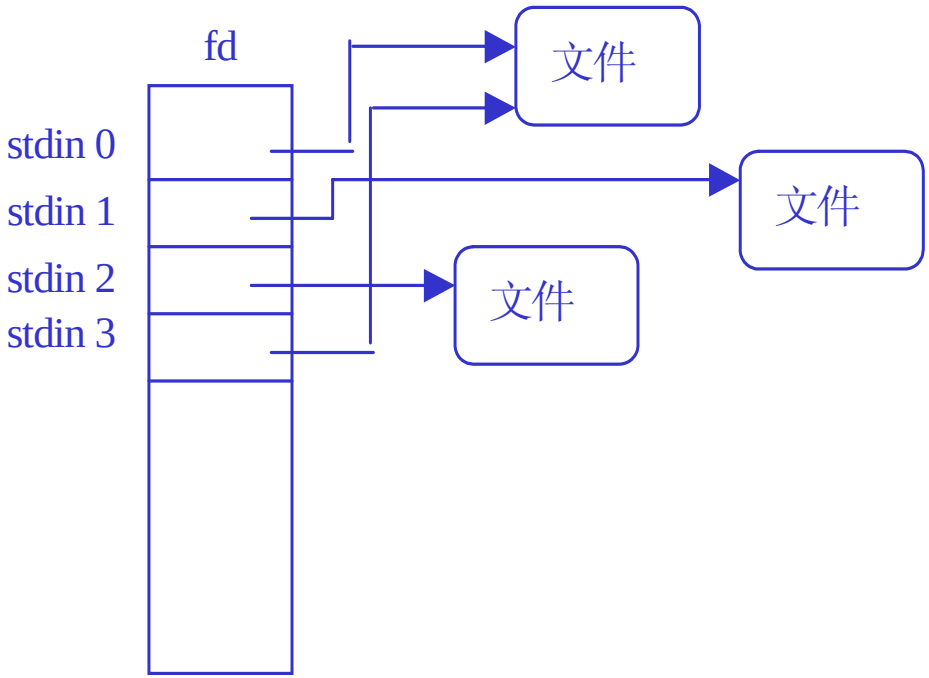


图8.5 文件描述符数组

与进程相关的文件结构 – fs_struct结构

- fs_struct结构描述进程与文件系统的关系

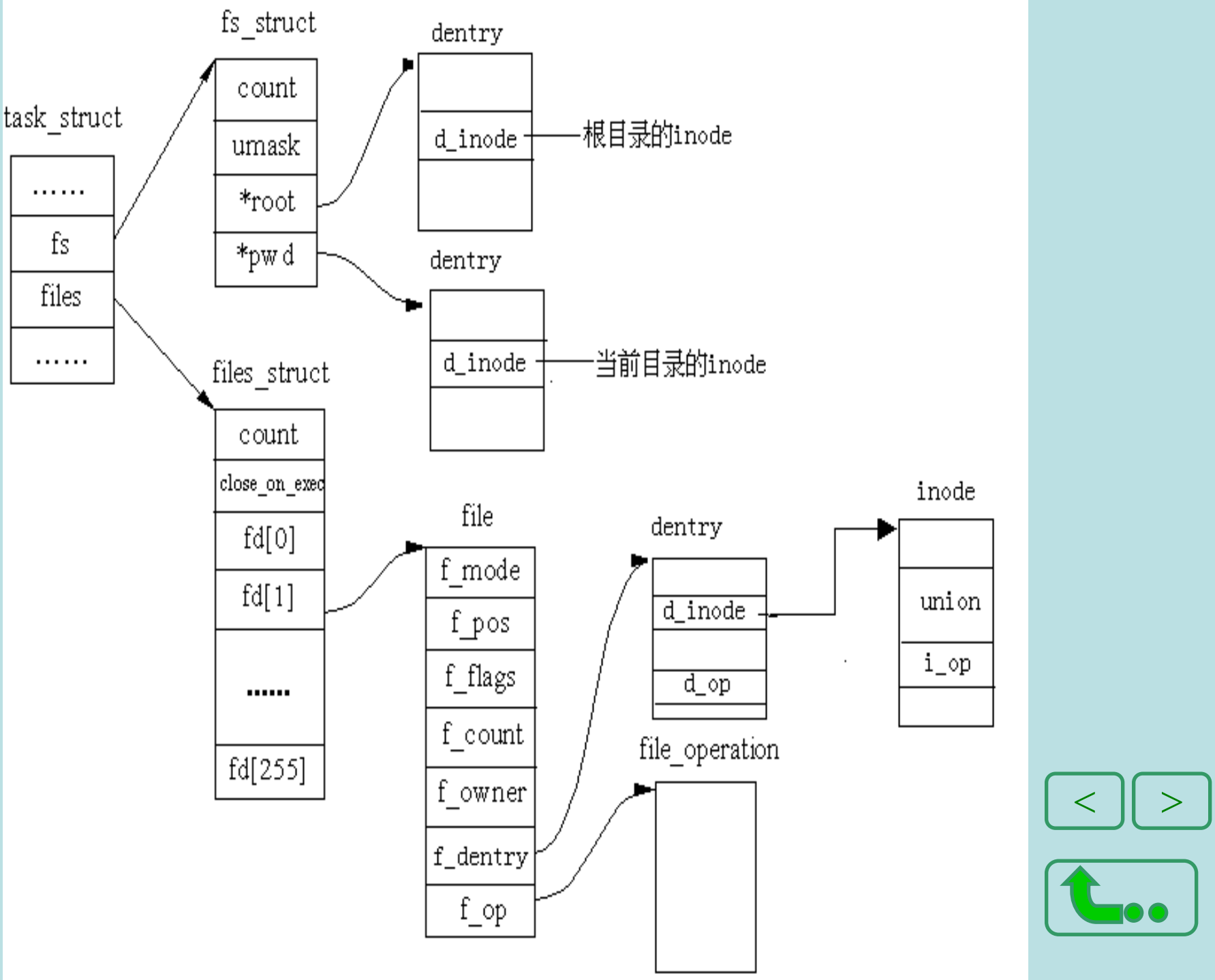
```
struct fs_struct {
   atomic_t count;
   rwlock_t lock;
   int umask;
   struct dentry * root,
        * pwd, * altroot;
   struct vfsmount
      * rootmnt, * pwdmnt,
      * altrootmnt;
};
```

count域表示共享同一fs_struct表的进程数目。umask域由umask（ ）系统调用使用，用于为新创建的文件设置初始文件许可权。

fs_struct中的dentry结构是对一个目录项的描述，root、pwd及 altroot三个指针都指向这个结构

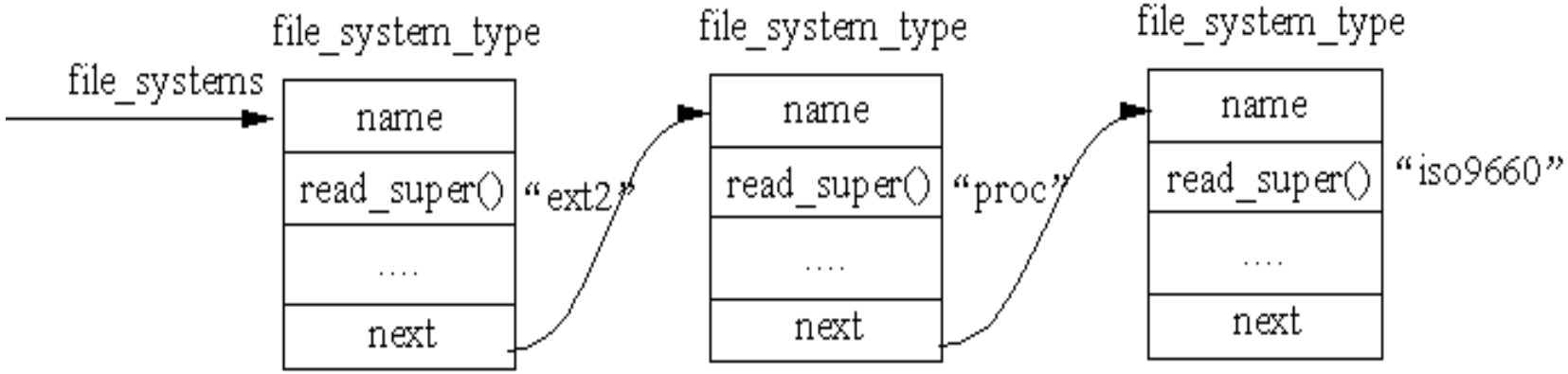
主要数据结构间的关系

- 超级块是对一个文件系统的描述；索引节点是对一个文件物理属性的描述；而目录项是对一个文件逻辑属性的描述
- 一个进程所处的位置是由fs_struct来描述的，而一个进程（或用户）打开的文件是由files_struct来描述的，而整个系统所打开的文件是由file结构来描述
- 主要数据结构间关系的图示


fs_struct
dentry
task_struct
count
umask
*root
*pwd
d_inode
根目录的inode
.....
fs
files
.....
dentry
d_inode
当前目录的inode
files_struct
count
close_on_exec
fd[0]
fd[1]
.....
fd[255]
file
inode
dentry
f_mode
f_pos
f_flags
f_count
f_owner
f_dentry
f_op
d_inode
d_op
union
i_op
file_operation

文件系统的注册和注销

- 当内核被编译时，就已经确定了可以支持哪些文件系统，这些文件系统在系统引导时，在 VFS 中进行注册。如果文件系统是作为内核可装载的模块，则在实际安装时进行注册，并在模块卸载时注销



文件系统的安装

- 安装一个文件系统实际上是安装一个物理设备
- 自己（一般是超级用户）安装文件系统时，需要指定三种信息：文件系统的名称、包含文件系统的物理块设备、文件系统在已有文件系统中的安装点。
- $ mount -t iso9660 /dev/hdc /mnt/cdrom 其中，iso9660是光驱文件系统的名称，/dev/hdc是包含文件系统的物理块设备，/mnt/cdrom就是将要安装到的目录，即安装点。
- 在用户程序中要安装一个文件系统则可以调用mount（）系统调用。安装过程主要工作是创建安装点对象，将其挂接到根文件系统的指定安装点下，然后初始化超级块对象，从而获得文件系统基本信息和相关的操作。

文件系统的卸载

- 如果文件系统中的文件当前正在使用，该文件系统是不能被卸载的
- 否则，查看对应的 VFS 超级块，如果该文件系统的 VFS 超级块标志为“脏”，则必须将超级块信息写回磁盘
- 之后，对应的 VFS 超级块被释放，vfsmount 数据结构将从vfsmntlist 链表中断开并被释放
- 具体的实现代码为fs/super.c中的sys_umount（）函数

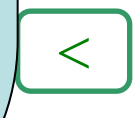
页缓冲区

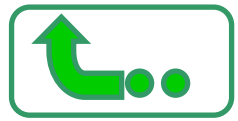
- 页缓冲区是由内存中的物理页组成的，缓冲区中每一页都对应着磁盘中的多个块
- 页缓存中的页来自读写常规文件、块设备文件和内存映射文件
- address space对象
- address space对象的操作函数表

Linux页缓冲区使用address_space结构描述页缓冲区中的页面：

```
struct address_space {
        struct inode                    *host;          /*属主的索引节点*/
        struct list_head        clean_pages;    /*干净页面链表*/
                struct list_head        dirty_pages;    /*脏页面链表*/
        struct list_head        locked_pages;   /*锁定页面链表*/
        struct list_head        io_pages; /*I/O使用页面链表*/
        unsigned long                   nrpages;  /*页面总数*/
        struct address_space_operations *a_ops;         /*操作表*/
        struct list_head        i_mmap;         /*私有映射链表*/
        struct list_head        i_mmap_shared;  /*共享映射链表*/
        struct semaphore        i_shared_sem;   /*保护上述两个链表*/
        unsigned long                   dirtied_when;   /*最后修改时间*/
        int                     gfp_mask;/*页面分配器的标志*/
        struct backing_dev_info *backing_dev_info; /*预读信息*/
        spinlock_t              private_lock;   /*私有address_space锁*/
        struct list_head        private_list;   /*私有address_space链表*/
        struct address_space            *assoc_mapping;  /*相关的缓存*/
};
```

address_space 对象

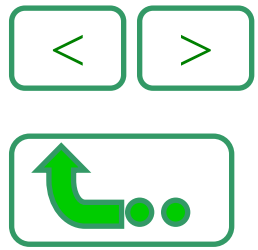




address_space对象的操作函数表

- a_ops域指向地址空间对象中的操作函数表，这与VFS对象和其操作表关系类似，操作函数表由address_space_operations结构表示

```
struct address_space_operations {
writepage()        /*写操作（把页写入磁盘）*/
readpage()         /*读操作（从磁盘读入页）*/
sync_page()
   /*启动在页上已经安排的I /O操作传送数据*/
prepare_write()     /*准备写操作*/
commit_write()      /*完成写操作*/
bmap()  /*从文件块索引获得逻辑块号*/
flushpage()        /*准备删除来自磁盘的页*/
releasepage()      /*由日志文件系统用来准备释放页*/
direct_IO()        /*数据页的直接I/O传送*/
}
```



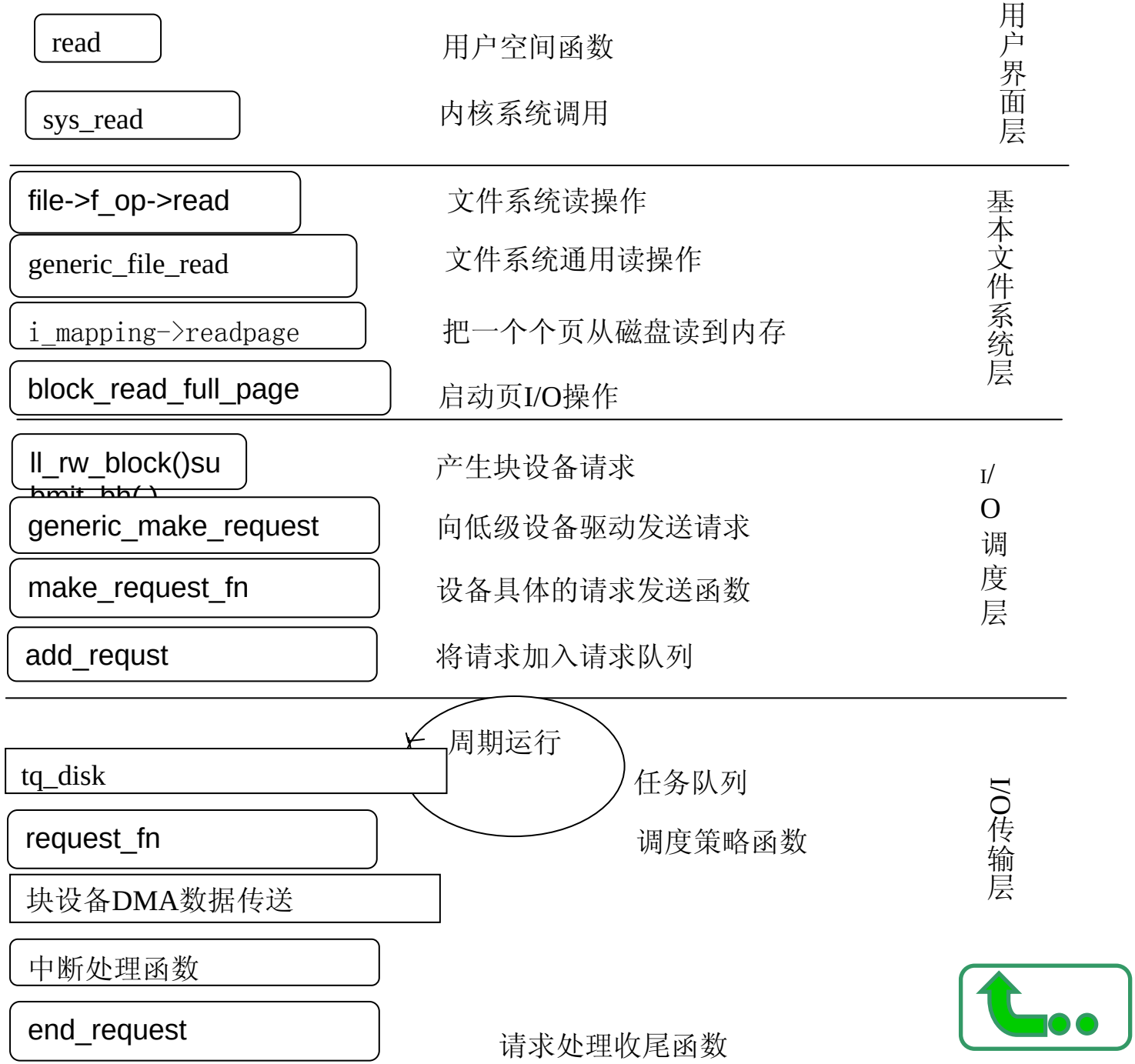
文件的打开

- open()系统调用就是打开文件，它返回一个文件描述符。
- 所谓打开文件实质上是在进程与文件之间建立起一种连接，而“文件描述符”唯一地标识着这样一个连接
- 打开文件，还意味着将目标文件的索引节点从磁盘载入内存，并进行初始化。
- 打开文件后，文件相关的“上下文”、索引节点、目录对象等都已经生成，

文件的读写

- 文件读写的基本步骤：
  - file=fget(fd)，也就是调用fget( )从fd获取相应文件对象的地址file，并把引用计数器file->f_count加1。
  - 检查file->f_mode中的标志是否允许所请求的访问（读或写操作）
  - 调用locks_verify_area( )检查对要访问的文件部分是否有强制锁
  - 调用file->f_op->read 或file->f_op->write来传送数据。这两个函数都返回实际传送的字节数。另一方面的作用是，文件指针被更新
  - 调用fput( )以减少引用计数器file->f_count的值。
  - 返回实际传送的字节数

读操作流程


read
用户空间函数
sys_read
内核系统调用
用户界面层
file->f_op->read
文件系统读操作
generic_file_read
文件系统通用读操作
i_mapping->readpage
把一个个页从磁盘读到内存
block_read_full_page
启动页I/O操作
基本文件系统层
ll_rw_block()su
bmit_bh( )
产生块设备请求
generic_make_request
向低级设备驱动发送请求
make_request_fn
设备具体的请求发送函数
add_requst
将请求加入请求队列
I/O调度层
周期运行
tq_disk
任务队列
request_fn
调度策略函数
块设备DMA数据传送
中断处理函数
end_request
请求处理收尾函数
I/O传输层

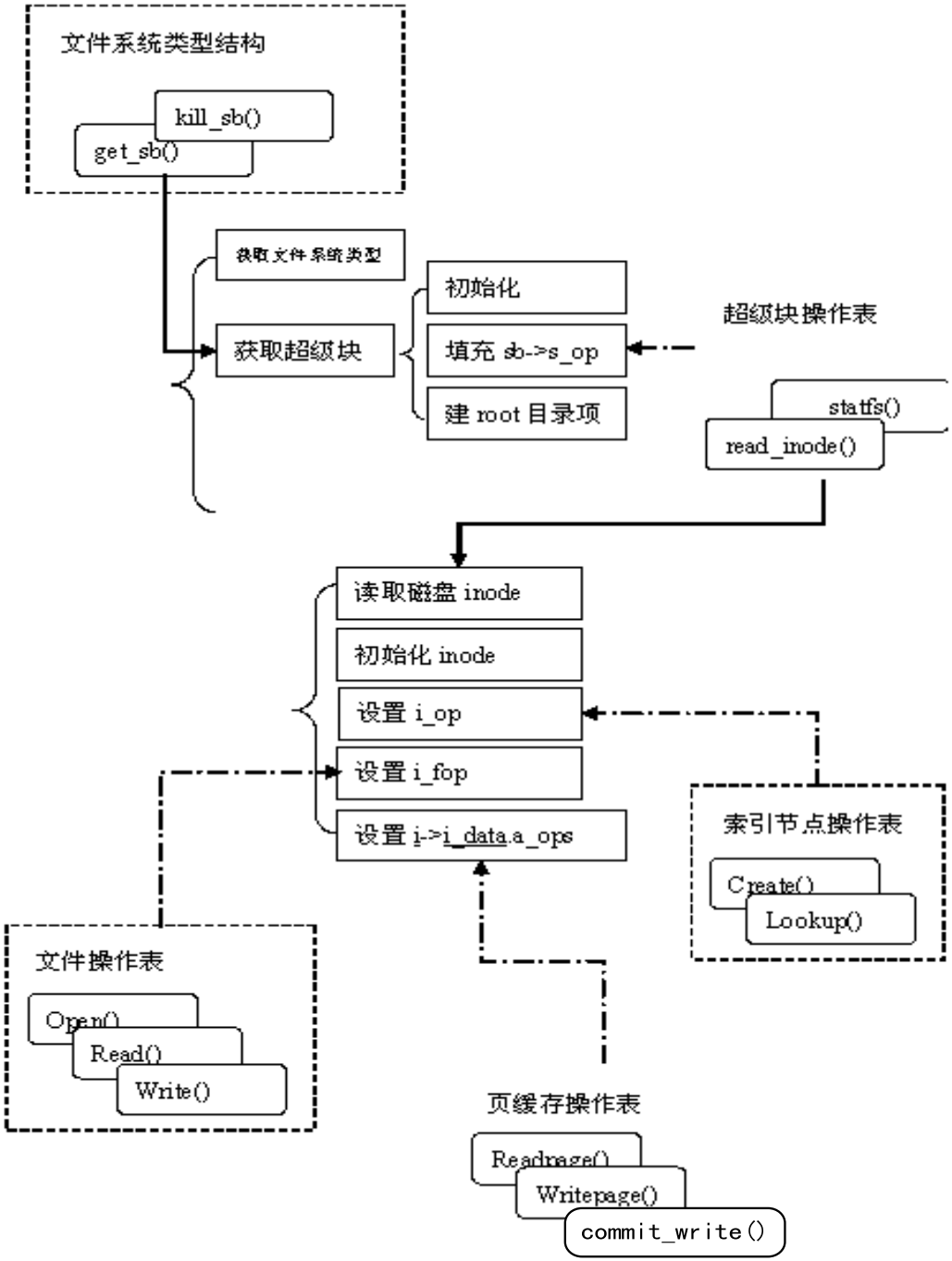
编写一个文件系统

- 文件系统比较庞杂，内核中提供的romfs文件系统是个非常理想的实例，我们以此为实例分析文件系统的实现。
- Linux文件系统的实现要素
- Romfs文件系统布局与文件结构
- 具体实现的对象

Linux文件系统的实现要素

- 编写新文件系统涉及一些基本对象，具体地说，需要建立“一个结构四个操作表”：
  - 文件系统类型结构（file_system_type）
  - 超级块操作表（super_operations）
  - 索引节点操作表（inode_operations）
  - 页缓冲区表（address_space_operations）
  - 文件操作表（file_operations）

一个结构及四个操作表之间的关系



Linux文件系统的实现要素

- 必须建立一个文件系统类型(file_system_type)来描述文件系统，它含有文件系统的名称、类型标志以及get_sb()等操作
- 超级块是我们寻找索引节点的唯一源头
- 索引节点需要许多自操作函数，这些函数都包含在索引节点操作表中
- 页缓冲区提供了页缓冲区操作表（address_space_operations），其中包含有readpage()、writepage()等函数负责对页缓冲区中的页进行读写等操作。

什么是Romfs文件系统

- Romfs是基于块的只读文件系统，它使用块（或扇区）访问存储设备
- 由于Romfs小型、轻量，所以常常用在嵌入系统和系统引导时
- Romfs是种很简单的文件系统，它的文件布局比Ext2等文件系统要简单得多
- Romfs比ext2文件系统需要更少的代码，且相对简单，建立文件系统超级块（superblock）需要更少的存储空间。

Romfs文件系统布局与文件结构

- 文件系统就是数据的分层存储结构。
- 在Linux内核源代码的Document/fs/romfs中介绍了romfs文件系统的布局和文件结构

0	文件系统名称
8	文件系统大小
12	检验和（前512字节）
16	卷名
	第一个文件头

Romfs文件系统布局

0	下一个文件头的偏移
4	文件类型
8	文件大小
12	检验和
16	文件

Romfs的文件结构

具体实现的对象

- 针对文件系统布局和文件结构， Romfs文件系统定义了一个磁盘超级块结构和文件的inode结构：

磁盘超级块结构：

```
struct romfs_super_block
 {
        _u32   word0;
        _u32   word1;
        _u32   size;
                _u32 checksum;
        char name[0];
 };
```

文件的inode结构：

```
 struct romfs_inode
{
        _u32 next;
        _u32 spec;
        _u32 size;
        _u32 checksum;
                char name[0];
};
```

“内核之旅”网站

- http://www.kerneltravel.net/
- 第七期“如何实现Linux下的文件系统”分析在Linux系统中如何实现新的文件系统。
- 在实例部分，我们将以romfs文件系统作实例分析实现文件系统的普遍步骤。
- 下载代码进行调试

第九章 设备驱动

- 设备驱动概述
- 设备驱动程序
- 专用I/O端口
- 字符设备驱动程序
- 块设备驱动程序

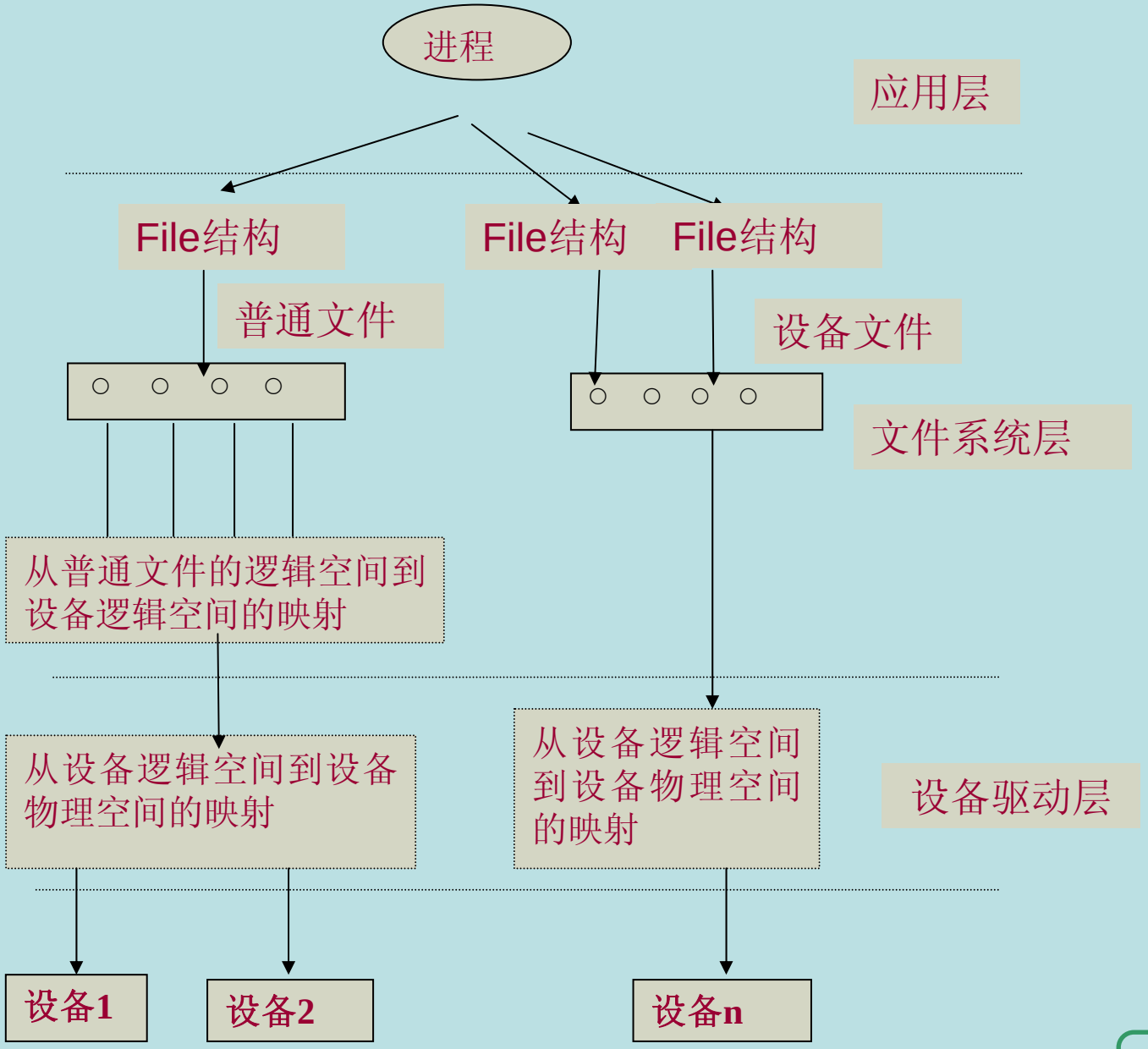
设备驱动概述

计算机中三个最基本的物质基础是CPU、内存和输入输出（I/O）设备，文件操作是对设备操作的组织和抽象，而设备操作则是对文件操作的最终实现

- Linux操作系统把设备纳入文件系统的范畴来管理
- 每个设备都对应一个文件名，在内核中也就对应一个索引节点
- 对文件操作的系统调用大都适用于设备文件
- 从应用程序的角度看，设备文件逻辑上的空间是一个线性空间（起始地址为0，每读取一个字节加1）。从这个逻辑空间到具体设备物理空间（如磁盘的磁道、扇区）的映射则是由内核提供，并被划分为文件操作和设备驱动两个层次

设备驱动概述

- 对于一个具体的设备而言，文件操作和设备驱动是一个事物的不同层次。从这种观点出发，从概念上可以把一个系统划分为应用、文件系统和设备驱动三个层次
- Linux将设备分成两大类。一类是像磁盘那样以块或扇区为单位，成块进行输入/输出的设备，称为**块设备**；另一类像键盘那样以字符（字节）为单位，逐个字符进行输入/输出的设备，称为**字符设备**。
- 文件系统通常都建立在块设备上


进程
应用层
File结构
File结构
File结构
普通文件
设备文件
文件系统层
从普通文件的逻辑空间到设备逻辑空间的映射
从设备逻辑空间到设备物理空间的映射
从设备逻辑空间到设备物理空间的映射
设备驱动层
设备1
设备2
设备n

设备驱动概述

- 为什么要把繁杂的设备归为“块设备”和“字符设备”两大类？
- 代表着设备的索引节点中记载着与特定设备建立连接所需的信息：文件（包括设备）的类型、主设备号和次设备号
- 要使一项设备可以被应用程序访问，首先要在系统中建立一个代表此设备的设备文件，这是通过系统调用mknode()实现的。此外，更重要的是在设备驱动层要有这种设备的驱动程序。
- 设备驱动层直接与物理设备打交道，在实际的实现中则因系统的结构和具体设备的物理特性不同而有不同的驱动方式

设备驱动程序基础

- 设备驱动程序:处理和管理硬件控制器的软件
- Linux 内核的设备管理是由一组运行在特权级上，驻留在内存以及对底层硬件进行处理的共享库的驱动程序来完成的
- 设备管理的一个基本特征是设备处理的抽象性，即所有硬件设备都被看成普通文件，可以通过用操纵普通文件相同的系统调用来打开、关闭、读取和写入设备
- 系统中每个设备都用一种设备文件来表示

设备驱动程序基础

I/O端口

设备文件

中断处理

设备驱动程序框架

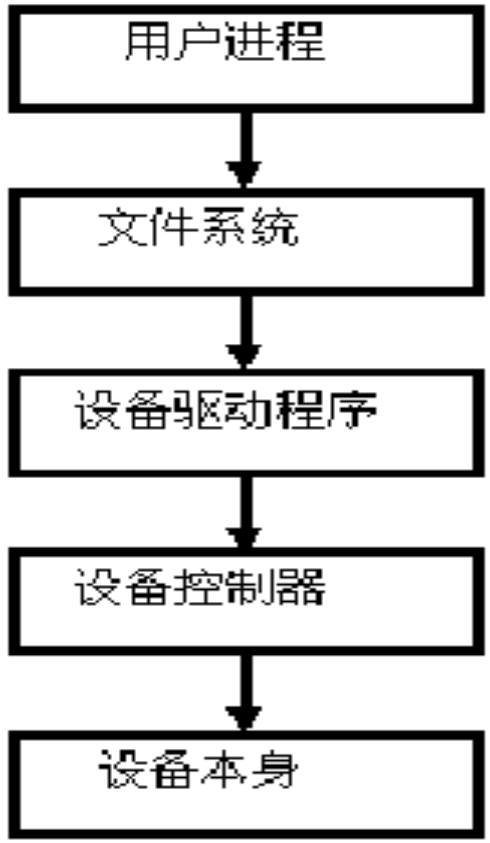


图9.2 用户进程请求设备服务的流程

I/O 端口

- 设备驱动程序要直接访问外设或其接口卡上的物理电路，通常以寄存器的形式出现
- 外设寄存器也称为I/O端口，通常包括控制寄存器、状态寄存器和数据寄存器三类
- “内存映射（Memory-mapped）”方式：
  - 寄存器参与内存统一编址，访问寄存器就通过访问一般的内存指令进行
- “I/O映射（I/O-mapped）”方式：
  - 将外设的寄存器看成一个独立的地址空间，为对外设寄存器的读/写设置专用指令

I/O 端口

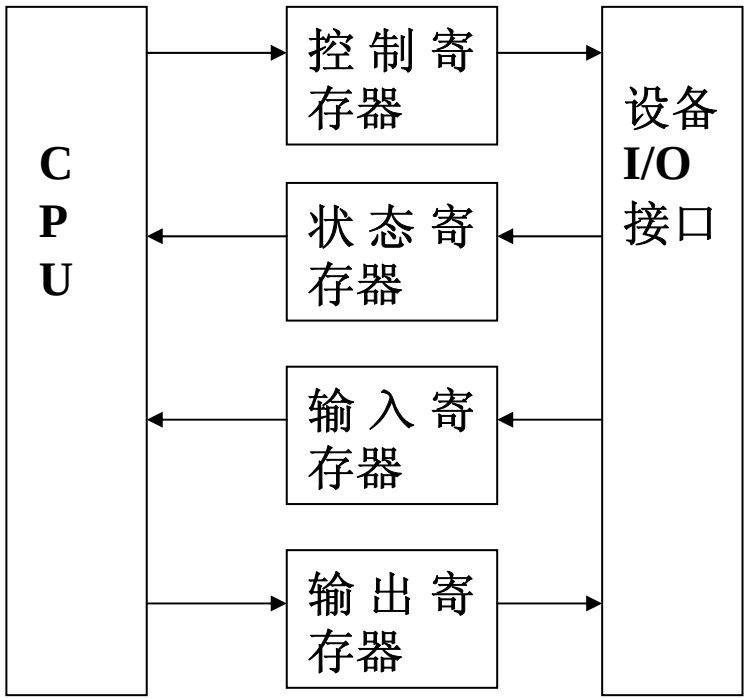


图9.3 专用I/O端口

- 系统设计者为了对I/O编程提供统一的方法，每个设备的I/O 端口都被组织成如图9.3所示的一组专用寄存器。
- 为了降低成本，通常把同一I/O端口用于不同目的
- 内核使用iotable表来记录分配给每个硬件设备的I/O端口

设备文件

- 设备文件用来表示Linux所支持的大多数设备，每个设备文件除了设备名外，还有类型、主设备号、次设备号这三个属性
- 设备文件是通过mknod系统调用创建的。其原型为：mknod(const char * filename, int mode, dev_t dev)
- 同一主设备号既可以标识字符设备，也可以标识块设备
- 一个设备文件通常与一个硬件设备相关联，或硬件设备的某一物理或逻辑分区。但有时，设备文件不会和任何实际的硬件关联，而是表示一个虚拟的逻辑设备。

中断处理

- 当设备执行某个命令时，设备驱动程序可以从查询方式和中断方式中选择一种来判断设备是否已经完成此命令
- 基于中断的设备驱动程序，指的是在硬件设备需要服务时向 CPU 发一个中断信号，引发中断服务处理程序执行
- 假定要实现一个简单的输入字符设备的驱动程序。当用户在相应的设备文件上发出read( )系统调用时，一条输入命令就发往设备的控制寄存器。在一个不可预知的长时间间隔后，设备把一个字节的数据放在输入寄存器。设备驱动程序返回这个字节作为read( )系统调用的结果。

设备驱动程序框架

- Linux的设备驱动程序与外设的接口可以分为三部分：
  - 驱动程序与内核的接口，这是通过数据结构file_operations来完成的。
  - 驱动程序与系统引导的接口，这部分利用驱动程序对设备进行初始化。
  - 驱动程序与设备的接口，这部分描述了驱动程序如何与设备进行交互，这与具体设备密切相关

设备驱动程序框架

- 根据功能，驱动程序的代码可以分为如下几个部分：
  - （1）驱动程序的注册和注销
  - （2）设备的打开与释放
  - （3）设备的读和写操作
  - （4）设备的控制操作
  - （5）设备的中断和查询处理
- 关于设备的控制操作可以通过驱动程序中的ioctl()来完成。与读写操作不同，ioctl()的用法与具体设备密切相关

字符设备驱动程序

- 传统的Unix设备驱动是以主／次设备号为主的，每个设备都有唯一的主设备号和次设备号，而内核中则有块设备表和字符设备表，根据设备的类型和主设备号便可在设备表中找到相应的驱动函数，而次设备号则一般只用作同类型设备中具体设备项的编号
- 由于字符设备的多样性，有时候也用次设备号作进一步的归类
- 字符设备驱动程序的注册

字符设备驱动程序的注册

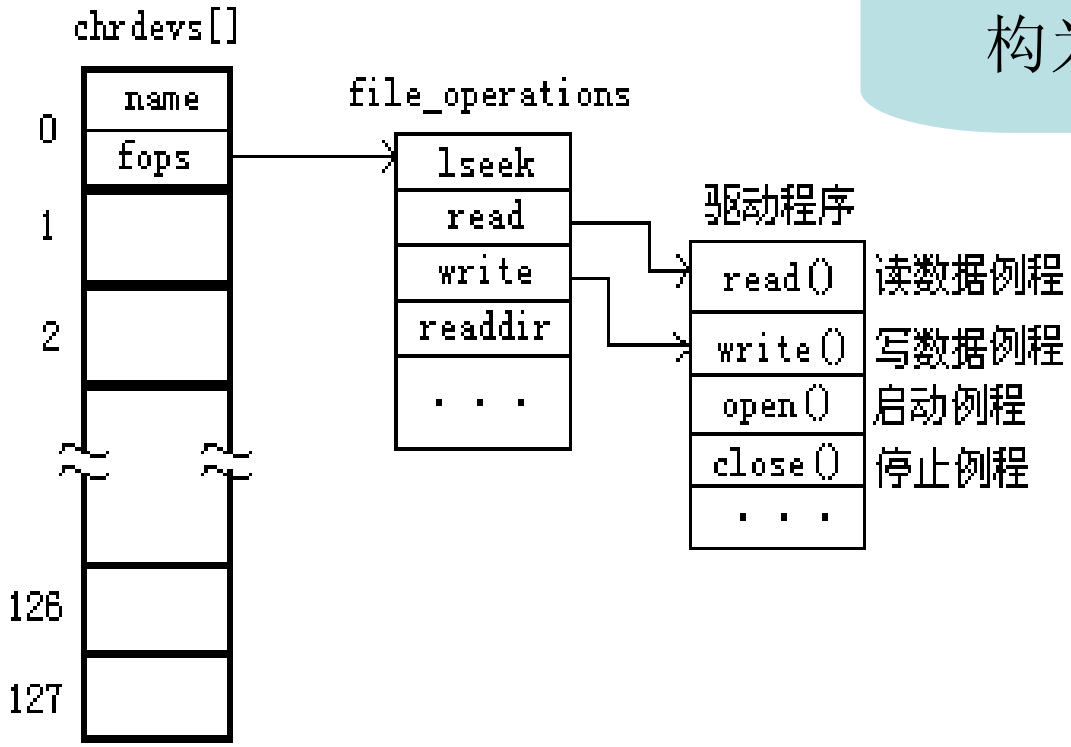


图9.4

字符设备注册表

用于注册字符设备的数据结构为device_struct：

```
struct device_struct
{
const char * name;
struct file_operations
 * fops;
};
```

简单字符设备驱动程序举例

- 驱动程序的编译、运行、安装及测试：
  - 使用makefile文件编译驱动程序：
    - 在命令行输入“ make -f test.mk”，则驱动程序被编译成一个可装载的“内核模块”。如果编译成功，则把它安装到系统中去
  - 安装和卸载驱动程序：
    - $ insmod -f chardev.o
    - $ rmmod test
  - 创建设备文件
    - $ mknod /dev/test c major minor
  - 编写测试程序

块驱动程序

- 块驱动程序提供了对面向块的设备的访问，这种设备以随机访问的方式传输数据，并且数据总是具有固定大小的块。典型的块设备是磁盘驱动器
- 块设备和字符设备的区别：
  - 块设备上可以mount文件系统，而字符设备是不可以的
  - 数据经过块设备相比操作字符设备需要多经历一个**数据缓冲层**（buffer cache mechanism）

块驱动程序的注册

```
struct block_device_operations
{
   int (*open) (struct inode *, struct file *);
            /*打开块设备文件*/
 int (*release) (struct inode *, struct file *);
/*关闭对块设备文件的最后一个引用 */
int (*ioctl) (struct inode *, struct file *,
unsigned, unsigned long);
/*在块设备文件上发出ioctl()系统调用*/
 int (*check_media_change) (kdev_t);
/*检查介质是否已经变化（如软盘）*/
 int (*revalidate) (kdev_t);
/*检查块设备是否持有有效数据*/
 };
```

内核使用主设备号来标识块驱动程序，但块主设备号和字符主设备号是互不相干的，它们具有各自独立的主设备号分配空间。

register_blkdev ( )使用block_device_operations结构的指针，该结构就是块驱动程序接口

注册和注销一个驱动程序模块时所要调用的函数及数据结构

- 加载模块时，insmod命令调用init_module()函数，该函数调用register_blkdev()和blk_init_queue()分别进行驱动程序的注册和请求队列的初始化。
- register_blkdev() 把块驱动程序接口block_device_operations加入blkdevs[]表中。
- blk_init_queue()初始化一个默认的请求队列，将其放入blk_dev[]表中，并将该驱动程序的 request 函数关联到该队列。
- 卸载模块时，rmmod命令调用cleanup_module()函数，该函数调用unregister_blkdev()和blk_cleanup_queue()分别进行驱动程序的注销和请求队列的清除

块设备请求

- 在内核安排一次数据传输时，首先在一个表中对该请求排队，并以最大化系统性能为原则进行排序。然后，请求队列被传到驱动程序的 request 函数
- 块设备的读写操作都是由request()函数完成，对于具体的块设备，函数request()是不同的。
- 所有的读写请求都存储在request结构的链表中
- request()函数从INIT_REQUEST宏命令开始，它对请求队列进行检查，保证请求队列中至少有一个请求在等待处理。如果没有请求，INIT_REQUEST宏命令将使request()函数返回，任务结束

块设备请求

- 块设备驱动程序初始化时，由驱动程序的init()完成。为了引导内核时调用init()，需要在blk_dev_init()函数中增加一行代码Mysdd_init()。
- 块设备驱动程序初始化的工作主要包括：
  - （1）检查硬件是否存在；
  - （2）登记主设备号；
  - （3）利用register_blkdev()函数对设备进行注册；
  - （4）将块设备驱动程序的数据容量传递给缓冲区

驱动程序—内核级程序开发的场所

- 在网上，Linux驱动程序的开发有大量的资料
- 《Linux Device Driver》是最精典的参考书
- 读者自己开发一个鼠标驱动程序。