“菜鸟玩转嵌入式”视频培训讲座
— Linux驱动开发基础班

主办：上海申嵌信息科技有限公司
承办：嵌入式家园
协办：上海嵌入式家园-开发板商城
广州友善之臂计算机科技有限公司
主讲：贺光辉（嵌入式系统工程师）

嵌入式家园 www.embedclub.com

上海嵌入式家园-开发板商城 http://embedclub.taobao.com/

上章回顾

- 中断处理程序架构
- 中断处理例程
- 中断顶半部与底半部
- 计时、延时函数
- 内核定时器与超时处理函数

第6章

内存与I/O访问

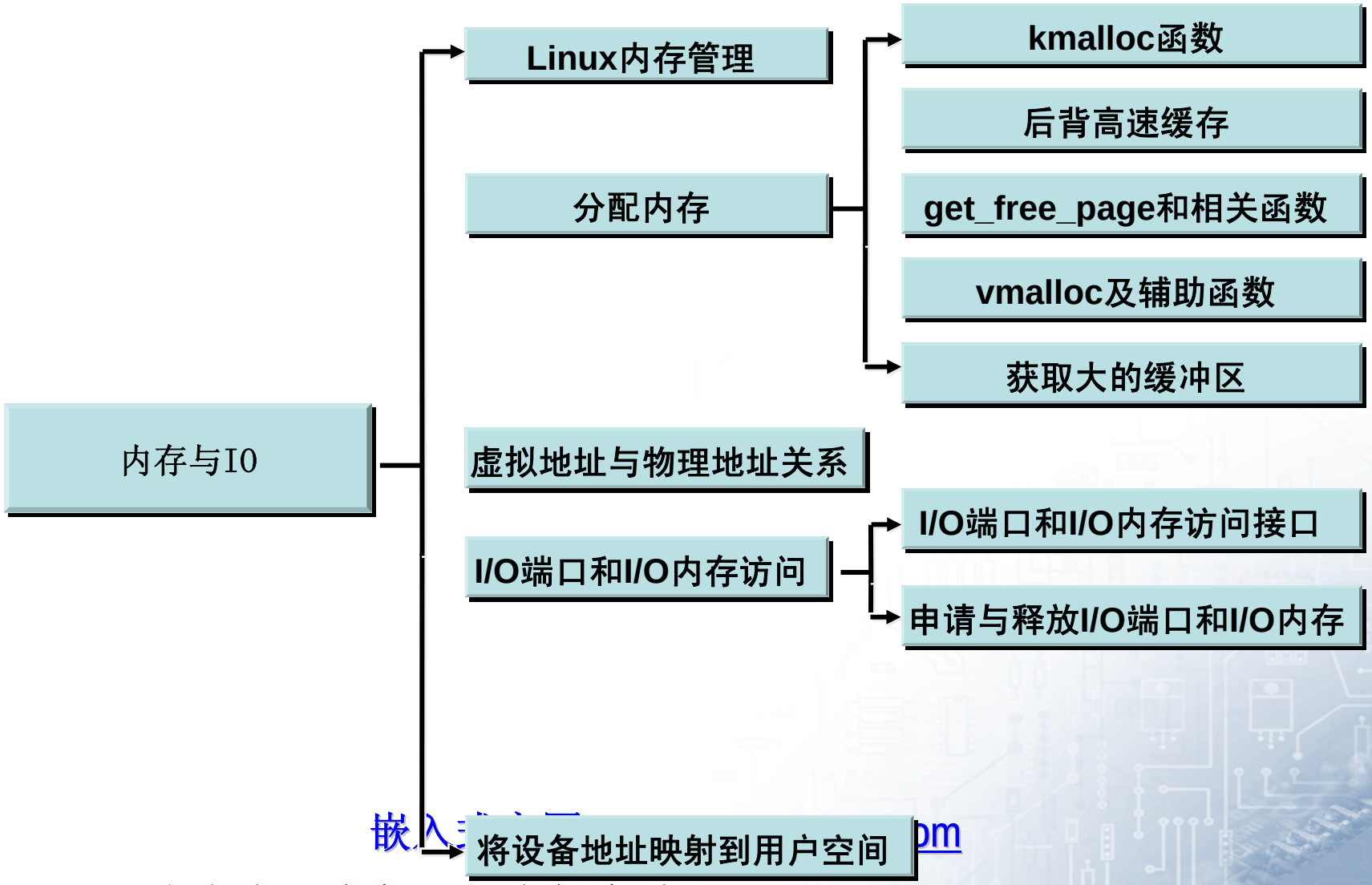
预习检查

- 什么是物理地址
- 什么是虚拟地址
- 简述物理地址与虚拟地址的关系

本章目标

- 掌握Linux内存分配的常用方法及区别
- 掌握I/O端口和I/O内存访问流程

本章结构



6.1 Linux内存管理

- 地址类型
- 物理地址和页
- 内存映射和页结构
- 页表

6.1.1 Linux地址类型

- **用户虚拟地址**
  - 用户空间程序所能看到的常规地址
  - 每个进程都有自己的虚拟地址空间
- **物理地址**
  - 该地址在处理器和系统内存之间使用
- **总线地址**
  - 该地址在外围总线和内存之间使用
  - 它实现总线和主内存之间的重新映射
  - 通常它们与处理器使用的物理地址相同

6.1.1 Linux地址类型

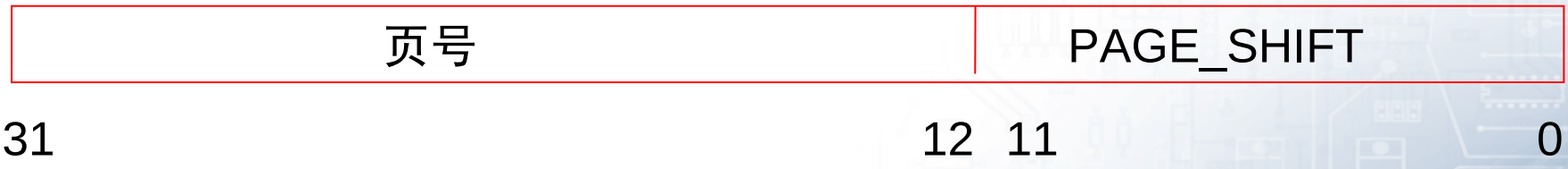
- **内核逻辑地址**
  - 内核逻辑地址组成了内核的常规地址空间。
  - 该地址映射了部分(或者全部)内存, 并经常被视为物理地址。
  - 与物理地址是线性映射（一一映射）的
  - 例如，kmalloc返回的是逻辑地址
- **内核虚拟地址**
  - 内核虚拟地址和逻辑地址的相同之处在于，它们都将内核空间的地址映射到物理地址上。
  - 与物理地址不必是线性映射关系
  - 例如，vmalloc与kmap都是返回内核虚拟地址
- **地址的转换**
  - __pa(logical-addr) //逻辑地址->物理地址
  - __va(physical-addr) //物理地址->逻辑地址

6.1.1 虚拟地址

- Linux操作系统采用虚拟内存管理技术，使得每个进程都有独立的进程地址空间，该空间的大小为3G,用户看到和接触的都是虚拟地址，无法看到实际的物理地址。利用这种虚拟地址不但能起到保护操作系统的作用，而且更重要的是用户程序可使用比实际物理内存更大的地址空间。
- Linux将4G的虚拟地址空间划分为两个部分： 用户空间和内核空间。
  - 用户空间：0x0 ~ 0xBFFFFFFF 即 0~3G
  - 内核空间：0xC0000000 ~ 0xFFFFFFFF 即 3G~4G
- 用户进程通常情况下只能访问到用户空间的虚拟地址，不能访问内核空间。例外情况是用户进程通过系统调用间接访问内核空间。

6.1.2 物理地址和页

- 物理地址被分成离散的大小相等单元，称之为页。
- Linux系统内部许多对内存的操作都是基于页的。
- 每个页的大小通常为4096个字节，具体的大小在<asm/page.h >中用PAGE_SIZE定义。
- 内存地址，无论是虚拟的还是物理的，它们都为分为页号和一个页内的偏移量。



- 页帧号：忽略地址偏移量，并将除去偏移量的剩余位移到右端，称该结果为页帧号。

6.1.3内存映射和页结构

- page的数据结构：struct page{…};
  - <linux/mm.h>
  - **atomic_t count;**
    - 对该页的访问计数。当计数值为0时，该页将返回给空闲链表。
  - **void *virtual;**
    - 如果页面被映射，则指向页的内核虚拟地址；如果未被映射则为NULL。
  - **unsigned long flags;**
    - 描述页状态的一系列标志.
    - PG_locked表示内存中的页已经被锁住
    - PG_reserved表示禁止内存管理系统访问该页。
- 内核维护了一个或者多个page结构数组，用来跟踪系统中的物理内存。

6.1.3内存映射和页结构

- page结构指针与虚拟地址之间进行转换

```
struct page *virt_to_page(void *kaddr)；//内核逻辑地址->page结构指针
struct page *pfn_to_page(int pfn)；//页帧号-> page结构指针
```

```
void *kmap(struct page *page)；//page结构指针->内核虚拟地址
void kunmap(struct page *page)；//释放映射
```

type参数指定使用哪个槽（专用页表入口）

```
//基于原子操作的kmap
void *kmap_atomic(struct page *page，enum km_type type)；
void kunmap_atomic(void *addr，enum km_type type)；
```


KM_USER0和KM_USER1(针对在用户空间中直接运行的代码)

KM_IRQ0和KM_IRQ1(针对中断处理程序)

6.1.4 页表

- 通常处理器必须使用某种机制，将虚拟地址转换为相应的物理地址，这种机制被称为页表。
- 页表是一个多层树形结构，结构化的数组中包含了虚拟地址到物理地址的映射和相关的标志位。
- 幸运的是，对驱动程序作者来说，在2.6版内核中删除了对页表直接操作的需求

6.2 分配内存

- 用户空间内存动态申请：
  - malloc/free
- 内核空间内存动态申请：
  - kmalloc()
  - __get_free_page()和相关函数
  - vmalloc()及其辅助函数

6.2.1 kmalloc函数

- kmalloc申请的内存位于物理内核映射区域，而且在物理上也是连续的，与真实物理地址只有一个固定的偏移。

```
#include<linux/slab.h>
void *kmalloc( size_t size, int flags );
```

分配的内存大小（字节数）

GFP_KERNEL：
由运行在内核态的进程调用，分配内存，可能引起睡眠

GFP_ATOMIC：
在中断处理函数、tasklet、内核定时器和持有自旋锁的时候申请内核内存，必须使用GFP_ATOMIC分配标志。

- GFP_KERNEL说明该内存分配是由运行在内核态的进程调用的。也就是说，调用它的函数是属于某个进程的，当空闲内存太少时，kmalloc函数会使当前进程进入睡眠，等待空闲页的出现。

6.2.2 后备高速缓存

- 设备驱动程序常常会反复地分配很多同一大小的内存块。为了满足这样的应用，内核实现了这种形式的内存池，通常称为后备高速缓存(lookaside cache) 。

6.2.3 __get_free_page和相关函数

- 如果模块需要分配大块的内存，那使用面向页的分配技术会更好

- 分配页面函数或宏

```
unsigned long get_zeroed_page(unsigned int flags);
// 返回指向新页面的指针，并将页面清零
unsigned long __get_free_page(unsigned int flags);
// 和get_zeroed_page类似，但不清零页面
unsigned long __get_free_pages(unsigned int flags，unsigned int order);
// 分配若干个连续的页面，返回指向该内存区域的指针，但也不清零这些内存区域
```

分配的页数为2^{order}

6.2.3 __get_free_page和相关函数

- 释放页面函数

分配的页数为2^{order}

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr，unsigned long order);
```

6.2.4 vmalloc及其辅助函数

- vmalloc分配虚拟地址空间的连续区域，但这段区域在物理上可能是不连续的。
- vmalloc不能用在原子上下文。因为它的内部实现使用了标志位GFP_KERNEL的kmalloc。

```
#include <linux/vmalloc.h>
void *vmalloc(unsigned long size);
void vfree(void *addr);
```

6.2.4 vmalloc及其辅助函数

- vmalloc分配得到的地址是不能在微处理器之外使用的，当驱动程序需要真正的物理地址时(像外设用以驱动系统总线的DMA地址)，就不能使用vmalloc；
- 使用vmalloc函数的正确场合是在分配一大块连续的、只在软件中存在的、用于缓冲的内存区域的时候。
- 因为vmalloc不但获取内存，还要建立页表，它的开销__get_free_pages 大，因此，用vmalloc函数分配仅仅一页的内存空间是不值得的。
- 通过vmalloc获取的内存使用起来效率不高，在大多数情况下不鼓励使用。尽可能直接与单个的页面打交道。

6.2.4 vmalloc及其辅助函数

I/O映射函数

```
void *ioremap(unsigned long offset，unsigned long size)；
void iounmap(void *addr)；
```

- 和vmalloc一样，ioremap也建立新的页表，但和vmalloc不同的是，ioremap并不实际分配内存。
- 使用ioremap()函数将设备所处的物理地址映射到虚拟地址。
- 为了保持可移植性，不应把ioremap返回的地址当作指向内存的指针而直接访问。相反，应该使用readb或其他I/O函数（完成设备内存映射的虚拟地址的读写）。

6.2.4 vmalloc及其辅助函数

- **vmalloc和kmalloc区别**
  - kmalloc使用的（虚拟）地址范围与物理内存是一一对应的；vmalloc使用的地址范围完全是虚拟的，每次分配都要通过对页表的适当设置来建立（虚拟）内存区域。 vmalloc申请的内存不一定是连续的。
  - vmalloc分配得到的地址是不能在微处理器之外使用的，因为它们只在处理器的内存单元上才有意义。当驱动程序需要真正的物理地址时(像外设用以驱动系统总线的DMA地址)，就不能使用vmalloc；
  - 通常，kmalloc分配小于128KB的内存，vmalloc可以分配更大的内存；
  - vmalloc不能在原子上下文中使用，因为它的内部实际调用了kmalloc(size, GFP_KERNEL)；

6.3 虚拟地址与物理地址关系

物理地址=虚拟地址-偏移量
（通常为3GB）

- 内核虚拟地址转化为物理地址

```
#define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
extern inline unsigned long virt_to_phys(volatile void *address)
{
        return __pa(address);
}
```

虚拟地址=物理地址+偏移量
（通常为3GB）

- 物理地址转化为内核虚拟地址

```
#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))
extern inline void * phys_to_virt(unsigned long address)
{
        return __va(address);
}
```

阶段总结

- Linux内存管理的一些基本概念
  - 地址类型
  - 页，页的结构，页表
- Linux内存分配的主要方法和区别
  - kmalloc
  - get_free_page
  - vmalloc等
- 虚拟地址与物理地址关系

6.4 I/O端口和I/O内存访问

- 设备通常会提供一组寄存器，如控制寄存器、数据寄存器和状态寄存器等。这些寄存器可能位于I/O空间，也可能位于内存空间。
- 当这些寄存器位于I/O空间时，通常被称为I/O端口
- 当这些寄存器位于内存空间时，对应的内存空间被称为I/O内存
- 通常, 除x86外（x86处理器提供了I/O空间），嵌入式处理器(比如ARM,PowerPC等)一般只存在内存空间。通过把I/O端口地址重新映射到内存地址来伪装端口I/O。eg:S3C2440平台下对I/O端口访问直接通过读写I/O内存来实现(eg:readb->inb)。

6.4.1 I/O端口和I/O内存访问接口

- I/O端口的操作（asm-generic/io.h）

```
unsigned inb(unsigned port); //读字节端口（8位宽）
void outb(unsigned char byte, unsigned port); //写字节端口（8位宽）

unsigned inw(unsigned port); //读字端口（16位宽）
void outw(unsigned short word, unsigned port); //写字端口（16位宽）

unsigned inl(unsigned port); //读长字端口（32位宽）
void outl(unsigned longword, unsigned port); //写长字端口（32位宽）

void insb(unsigned port, void *addr, unsigned long count);//读一串字节
void outsb(unsigned port, void *addr, unsigned long count);//写一串字节

void insw(unsigned port, void *addr, unsigned long count);//读一串字
void outsw(unsigned port, void *addr, unsigned long count);//写一串字

void insl(unsigned port, void *addr, unsigned long count) ;//读一串长字
void outsl(unsigned port, void *addr, unsigned long count);//写一串长字
```

6.4.1 I/O端口和I/O内存访问接口

- I/O 内存（asm-generic/io.h）
  - 在内核中访问I/O内存之前，需首先使用ioremap()函数将设备所处的物理地址映射到虚拟地址, ioremap()的原型如下(asm-generic/io.h)：

```
void * ioremap(unsigned long offset，unsigned long size);
```

  - 访问函数

addr为通过ioremap获取的地址

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
```

```
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

```
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

6.4.1 I/O端口和I/O内存访问接口

读一串I/O内存

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
```

写一串I/O内存

```
void iowrite8_rep(void *addr, const void *buf, unsigned long count)；
void iowrite16_rep(void *addr, const void *buf, unsigned long count)；
void iowrite32_rep(void *addr, const void *buf, unsigned long count)；
```

复制I/O内存

```
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

设置I/O内存

```
void memset_io(void *addr, u8 value, unsigned int count)；
```

6.4.2 申请与释放设备I/O端口和I/O内存

- I/O端口申请与释放

向内核登记需要使用起始于基地址为first的n个I/O端口。分配的I/O端口可从文件/proc/ioports中得到

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n,
                                const char *name);
void release_region(unsigned long start, unsigned long n);
```

使用这段I/O端口的设备的名称

- I/O内存的申请与释放

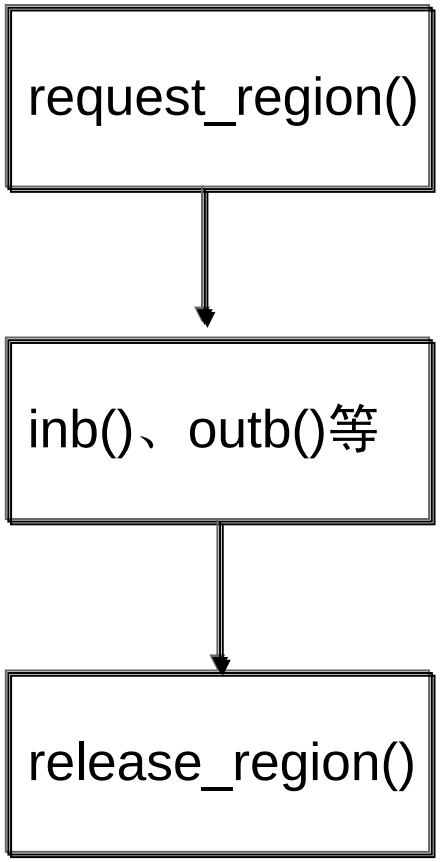
```
struct resource *request_mem_region(unsigned long start, char *name);
void release_mem_region(unsigned long start, unsigned long len);
```

- 上述request_region()和request_mem_region()都不是必须的，但建议使用。其任务是检查申请的资源是否可用，如果可用则申请成功，并标志为已经使用，其他驱动想再次申请该资源时就会失败。

6.4.2 I/O端口和I/O内存操作函数代码分析

- 【代码阅读】：
- linux-2.6.32.2/arch/arm/plat-s3c24xx/gpio.c
- linux-2.6. 32.2/include/asm-generic/io.h
- linux-2.6. 32.2/arch/arm/mach-s3c2410/include/mach/regs-gpio.h

6.4.2 I/O端口的访问流程

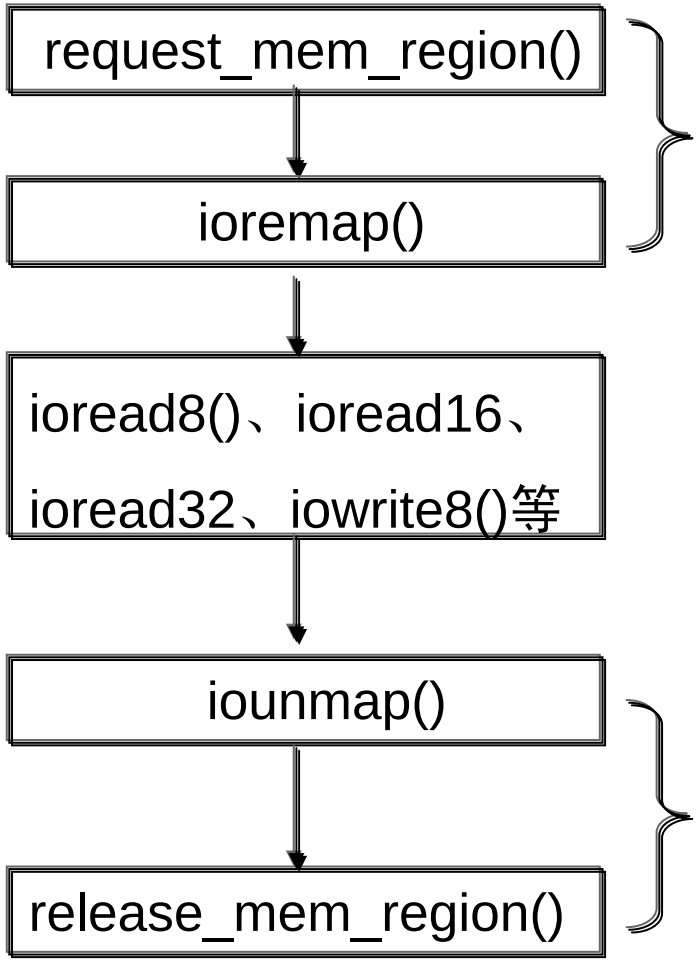


在设备驱动模块加载或open()函数中进行

在设备驱动初始化、read()、write()、ioctl()等函数中进行

在设备驱动模块卸载或release()函数中进行

6.4.2 I/O内存的访问流程



在设备驱动模块加载或open()函数中进行

在设备驱动初始化、read()、write()、ioctl()等函数中进行

在设备驱动模块卸载或release()函数中进行

6.5 将设备地址映射到用户空间

- 用户空间不能直接访问设备。
- 设备驱动程序中可实现mmap()函数，可使得用户空间能直接访问设备的物理地址
  - mmap()将用户空间的一段内存与设备内存关联
  - 当用户访问用户空间的这段地址范围时，会转化为对设备的访问
- mmap()必须以PAGE_SIZE为单位进行映射
  - 内存只能以页为单位进行映射
  - 并且进行页对齐，以PAGE_SIZE的倍数大小进行映射

6.5 将设备地址映射到用户空间

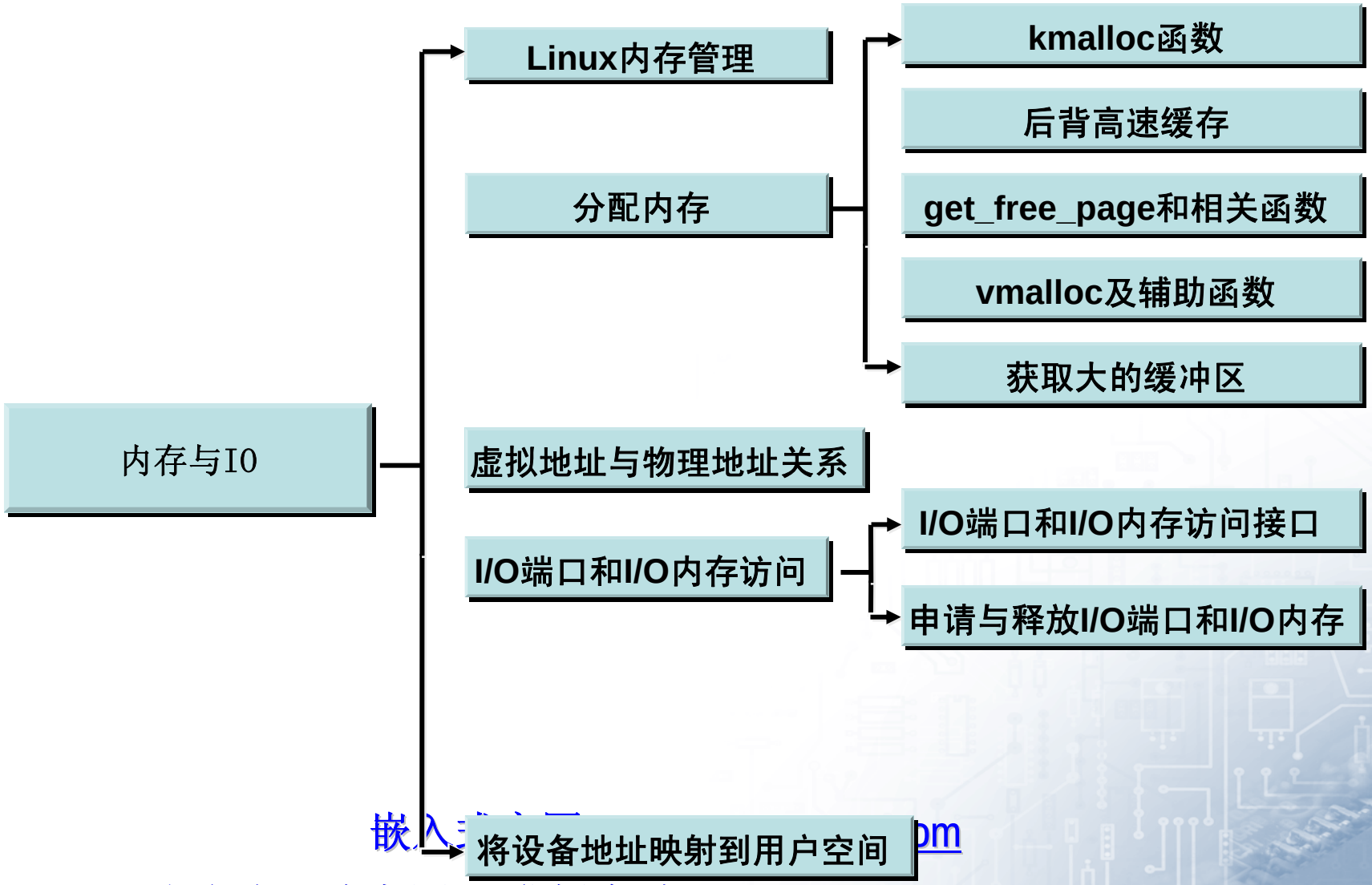
- mmap函数的原型

```
int (*mmap)(struct file *filp，struct vm_area_struct *vma)；
```

```
void(*open)(struct vm_area_struct *vma)；
void(*close)(struct vm_area_struct *vma)；
struct page *(*nopage)(struct
        vm_area_struct *vma，
        unsigned long address，
        int *type)；
int (*populate)(struct vm_area_struct
        *vm,unsigned long address，
        unsigned long len，
        pgprot_t prot，
        unsigned long pgoff，
        int nonblock)；
```

```
用于访问设备的虚拟地址的信息:
unsigned long vm_start；
unsigned long vm_end；
struct file *vm_file；
unsigned long vm_pgoff；
unsigned long vm_flags；
struct vm_operations_struct *vm_ops；
void *vm_pivate_data；
```

本章结构



- 任务1：内核空间内存动态申请实验（kmalloc, get_zeroed_page, vmalloc）
- 任务2：使用ioremap地址映射函数重写Beep驱动