**概念**

以下概念使用的示例基于 2g 内存条，32 位地址总线

了解以下概念需要先知道，除了内存管理子系统会直接使用物理地址外，其它程序都是使用虚拟地址

**物理地址：**

cpu把物理地址分为帧

Cpu可以访问的真正的地址，例如你的内存条只有2g，你不能说我要取物理地址在0xFFFFFFFF（4g位置）位置的值

**用户虚拟地址：**

每个进程有4g的虚拟地址，其中0-3g分给用户空间，这段就是用户虚拟地址

**内核虚拟地址：**

每个进程有4g的虚拟地址，其中0-3g分给用户空间，3g-4g则分给内核空间，这段空间就是内核虚拟地址，使用vmalloc、kmap返回的便是虚拟内存

**内核逻辑地址：**

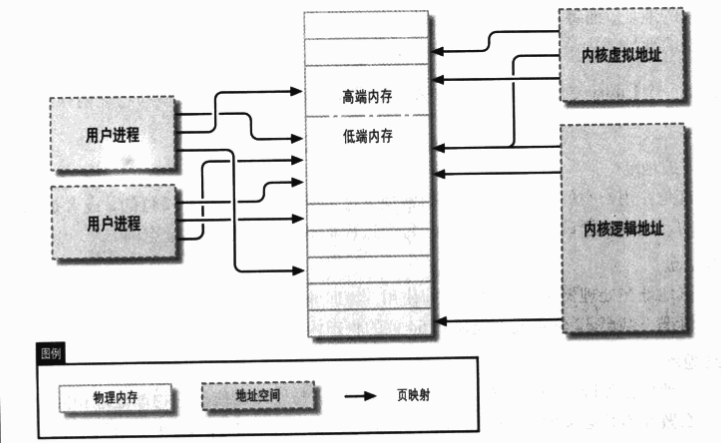
3g-4g为内核虚拟地址，但内核实际上将3g-3g896m的地址固定映射到内存的0-896m的地址上，3g-3g896m的地址称为逻辑地址，使用kmalloc分配的便是此地址

**低端内存：**

0-896m的内存为低端内存，其被固定映射到内核的逻辑地址

**高端内存：**

>896m的内存为高端内存，高端内存没有直接映射到虚拟地址上，为访问高端内存，内核使用页表保存 虚拟地址 到 高端内存地址 的映射



**物理地址和页**

物理地址被分为页，PAGE\_SIZE (定义在 <asm/page.h>) 给出了页大小，32位的页地址油 页号（20位）：偏移量（12位） 偏移量的位数可从PAGE\_SHIFT宏获取

**页映射与页结构**

系统中每个物理页都有一个page结构相对应

**Page结构**

atomic\_t count;

这个页的引用数. 当这个 count 掉到 0, 这页被返回给空闲列表.

void \*virtual;

如果页被映射，则指向页的虚拟地址，否则, NULL

unsigned long flags;

一套描述页状态的一套位标志.

**page结构指针与虚拟地址的转换**

struct page \*virt\_to\_page(void \*kaddr);

返回内核逻辑地址相对应的page结构的指针，这个宏定义在 <asm/page.h>

struct page \*pfn\_to\_page(int pfn);

为给定的页帧号pfn返回对应的 struct page 指针.

void \*page\_address(struct page \*page);

返回这个页page对应的内核虚拟地址，这个函数在 <linux/mm.h> 中定义. 大部分情况下, 你应使用 kmap 的而不是 page\_address.

#include <linux/highmem.h>

void \*kmap(struct page \*page);

void kunmap(struct page \*page);

kmap 为系统中的页返回一个内核虚拟地址

#include <linux/highmem.h>

#include <asm/kmap\_types.h>

void \*kmap\_atomic(struct page \*page, enum km\_type type);

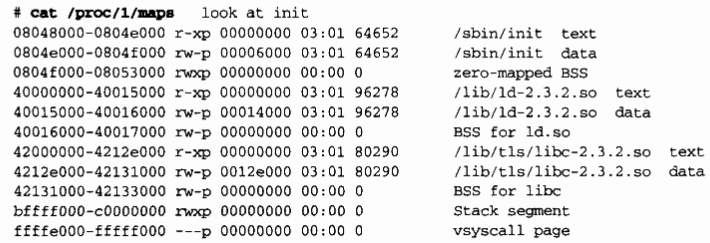
void kunmap\_atomic(void \*addr, enum km\_type type);

kmap\_atomic 是 kmap 的一种高性能形式.

**虚拟内存区（VMA）**

一个进程的可能有多个可执行代码区（text）、数据区（data）等

通过访问/proc/<pid/maps>（这里 pid 用一个进程的 ID 来替换）查看进程的虚拟内存区



其每行的格式为

start-end perm offset major:minor inode image

Start-end

这个内存区的开始和结束的虚拟地址.

perm

虚拟内存区的读,写和执行权限的位掩码

offset

内存区在它被映射到的文件中的起始位置

major minor

持有已被映射文件的设备的主次编号

inode

被映射文件的 inode 号.

image

被映射的文件名

**vm\_area\_struct 结构**

vm\_area\_struct 结构代表一个虚拟内存区（在 <linux/mm.h> 中定义）

unsigned long vm\_start;

unsigned long vm\_end;

虚拟内存区的开始和结束的虚拟地址

struct file \*vm\_file;

一个指向和这个区关联的 struct file 结构的指针.

unsigned long vm\_pgoff;

以页为单位，该区域在文件中的偏移

unsigned long vm\_flags;

描述这个区的一套标志. VM\_IO 标志一个 VMA 作为内存映射的 I/O 区，VM\_RESERVED 告知内存管理系统不要将这个VMA交换出去

struct vm\_operations\_struct \*vm\_ops;

用来对该内存区进行操作的函数集合

void \*vm\_private\_data;

驱动可以用来存储它的自身信息的成员.

**vm\_operations\_struct结构**

vm\_operations\_struct结构是VMA的操作集合

void (\*open)(struct vm\_area\_struct \*vma);

在VMA初始化时调用

void (\*close)(struct vm\_area\_struct \*vma);

当销毁一个区域时内核会调用它

struct page \*(\*nopage)(struct vm\_area\_struct \*vma, unsigned long address, int \*type);

当一个进程试图存取使用一个有效 VMA 的页, 但是它当前不在内存中, nopage 方法被调用. 这个方法返回 struct page 指针给物理页

int (\*populate)(struct vm\_area\_struct \*vm, unsigned long address, unsigned long len, pgprot\_t prot, unsigned long pgoff, int nonblock);

用户空间访问页前，该函数将页装入内存，驱动程序一般不实现该方法

**关于内存映射**

虚拟内存到物理内存的映射保存在页表中

每个进程都用一个struct mm\_struct（定义在 <linux/sched.h>），其中包含了虚拟内存区的链表，页表，通过current->mm可以访问mm\_struct

**mmap内存映射操作**

mmap操作用于将文件映射到虚拟地址上

**用户空间mmap函数**

void \*mmap(void \*start, size\_t length,int prot,int flags,int fd,off\_t offsize);

start：

指向欲映射的内存起始地址，通常设为 NULL，代表让系统自动选定地址，映射成功后返回该地址。

length：代表将文件中多大的部分映射到内存。

prot：映射区域的保护方式

flags：影响映射区域的各种特性。

fd：要映射到内存中的文件描述符。

offset：文件映射的偏移量，通常设置为0，offset必须是分页大小的整数倍。

返回值：若映射成功则返回映射区的内存起始地址，否则返回MAP\_FAILED(－1)

**驱动程序的mmap**

要支持mmap，驱动程序需要在file\_operation 结构中实现mmap，其原型如下

int (\*mmap) (struct file \*filp, struct vm\_area\_struct \*vma);

mmap的任务：

建立页表映射

为vma->vm\_ops指定虚拟内存区操作

**remap\_pfn\_range函数建立页表映射**

remap\_pfn\_range只能映射保留页和超出物理内存的地址，在pc中640kb-1m的地址为保留的，超出物理内存地址的即为设备内存

int remap\_pfn\_range(struct vm\_area\_struct \*vma, unsigned long virt\_addr, unsigned long pfn, unsigned long size, pgprot\_t prot);

vma

页范围被映射到的虚拟内存区

virt\_addr

要映射的虚拟地址的开始

pfn

要映射的物理地址的页帧号

size

要映射的大小, 以字节.

prot

VMA的保护属性

该函数使用的参数内核已经在vma中为我们指定好了

**一个简单的mmap示例：**

static int simple\_remap\_mmap(struct file \*filp, struct vm\_area\_struct \*vma)

{

    // 页表映射，在mmap中一次性映射所有所需的虚拟内存

    if (remap\_pfn\_range(vma, vma->vm\_start, vm->vm\_pgoff,

                        vma->vm\_end - vma->vm\_start,

                        vma->vm\_page\_prot))

        return -EAGAIN;

    // 为虚拟内存区指定操作

    vma->vm\_ops = &simple\_remap\_vm\_ops;

    // 显示调用 vma->ops->open 方法打开虚拟内存

    simple\_vma\_open(vma);

    return 0;

}

**nopage内存映射**

nopage函数定义在vm\_area\_struct的vm\_operations\_struct中

当进程试图当我的虚拟内存页没有对应的物理地址时，就会调用该函数，例如用户空间使用mremap系统调用重新映射虚拟内存，这时会有一部分虚拟地址没有对应的物理内存

struct page \*(\*nopage)(struct vm\_area\_struct \*vma, unsigned long address, int \*type);

Address：试图访问的虚拟地址

返回值：返回对应的物理页

**使用nopage内存映射的示例：**

使用nopage后，我们无需在mmap中以开始就映射所有虚拟内存

static int simple\_nopage\_mmap(struct file \*filp, struct vm\_area\_struct \*vma)

{

    // ......

    // 告知内存管理系统不要将这个VMA交换出去

vma->vm\_flags |= VM\_RESERVED;

    // 设置虚拟内存区操作

    vma->vm\_ops = &simple\_nopage\_vm\_ops;

    // 打开虚拟内存

    simple\_vma\_open(vma);

    return 0;

}

nopage函数需要返回page指针

struct page \*simple\_vma\_nopage(struct vm\_area\_struct \*vma, unsigned long address, int \*type)

{

    struct page \*pageptr;       // page 指针

    unsigned long pageframe;    // 需要分配的页帧号

// 获取一个有效的页帧号.... 如：pageframe = 10;

// 如果用户请求的mmap的fd是设备文件，则页帧号应该对应我们设备内存的页帧号

    // 验证页帧号是否有效

    if (!pfn\_valid(pageframe))

        return NOPAGE\_SIGBUS;

    // 获取页帧号对应的 page 指针

    pageptr = pfn\_to\_page(pageframe);

    // 增加 page 的引用

    get\_page(pageptr);

    if (type)

        \*type = VM\_FAULT\_MINOR;

    return pageptr;

}

**直接IO访问**

不做介绍

**直接内存访问（DMA）**

DMA允许外围设备和主存直接直接传输数据，不需要cpu参与

**判断当前设备是否能够执行DMA**

int dma\_set\_mask(struct device \*dev, u64 mask);

Mask：当前设备的寻址位，如当前设备寻址为 24 位, 例如, 你要传 0x0FFFFFF.

返回值：是非零如果使用给定的 mask 可以 DMA; 否则不能进程DMA操作

示例：

if (dma\_set\_mask(dev, 0xffffff))

    card->use\_dma = 1;

else

{

    card->use\_dma = 0; /\* We'll have to live without DMA \*/

    printk(KERN\_WARN, "mydev: DMA not supported\n");

}

**一致性DMA映射**

建立映射

void \*dma\_alloc\_coherent(struct device \*dev, size\_t size, dma\_addr\_t \*dma\_handle, int flag);

Dev：设备

Size：缓存区大小

Dma\_handle：作为返回值，保存总线地址，无需了解其中的含义

Flag：内存分配标记，GFP\_KERNEL 或者 GFP\_ATOMIC

返回值：以映射到的缓存区的内核虚拟地址

释放映射

void dma\_free\_coherent(struct device \*dev, size\_t size, void \*vaddr, dma\_addr\_t dma\_handle);

DMA池

通过dma\_alloc\_coherent建立的最小映射为单页，如果需要更新的映射，则应使用DMA池

创建DMA池

struct dma\_pool \*dma\_pool\_create(const char \*name,

                                 struct device \*dev,

                                 size\_t size,

                                 size\_t align,

                                 size\_t allocation);

Name：dma池的名字

Dev：设备

Size：从池中分配的缓冲区的大小

Align：硬件对齐原则

Allocation：如果不为0，则表示内存边界不能超过allocation

释放DMA池

void dma\_pool\_destroy(struct dma\_pool \*pool);

DMA池中获取缓存区

void \*dma\_pool\_alloc(struct dma\_pool \*pool, int mem\_flags, dma\_addr\_t \*handle);

Mem\_flags：kmalloc内存分配标志

Handle：返回的总线地址（无需关心）

返回值：缓存区的指针

释放DMA池的缓存区

void dma\_pool\_free(struct dma\_pool \*pool, void \*vaddr, dma\_addr\_t addr);

**流式DMA映射**

创建缓存区映射

dma\_addr\_t dma\_map\_single(struct device \*dev, void \*buffer, size\_t size, enum dma\_data\_direction direction);

Direction：数据传输方向

DMA\_TO\_DEVICE ：往设备的数据

DMA\_FROM\_DEVICE ：来自设备的数据

DMA\_BIDIRECTIONAL ：双向

DMA\_NONE ：作为调试辅助标志

删除映射

void dma\_unmap\_single(struct device \*dev, dma\_addr\_t dma\_addr, size\_t size, enum dma\_data\_direction direction);

**一个简单的DMA映射示例**

int dad\_transfer(struct dad\_dev \*dev, int write, void \*buffer, size\_t count)

{

    dma\_addr\_t bus\_addr;

    /\* 映射设备到缓存 \*/

    dev->dma\_dir = (write ? DMA\_TO\_DEVICE : DMA\_FROM\_DEVICE);

    dev->dma\_size = count;

    bus\_addr = dma\_map\_single(&dev->pci\_dev->dev, buffer, count, dev->dma\_dir);

    dev->dma\_addr = bus\_addr;

    /\* 设置设备 \*/

    writeb(dev->registers.command, DAD\_CMD\_DISABLEDMA);

    writeb(dev->registers.command, write ? DAD\_CMD\_WR : DAD\_CMD\_RD);

    writel(dev->registers.addr, cpu\_to\_le32(bus\_addr));

writel(dev->registers.len, cpu\_to\_le32(count));

    /\* 开始操作 \*/

    writeb(dev->registers.command, DAD\_CMD\_ENABLEDMA);

    return 0;

}

中断例程

void dad\_interrupt(int irq, void \*dev\_id, struct pt\_regs \*regs)

{

    struct dad\_dev \*dev = (struct dad\_dev \*)dev\_id;

/\* 确定中断设备是从对应的设备发来的 \*/

    /\* 解除DMA映射 \*/

dma\_unmap\_single(dev->pci\_dev->dev, dev->dma\_addr, dev->dma\_size, dev->dma\_dir);

    /\* 访问缓存区 \*/

    // ...

}