

Tekkaman Ninja

tekkamanninja.blog.chinaunix.net

Linux我的梦想，我的未来！ 本博客的原创文章的内容会不定期更新或修正错误！转载文章都会注明出处，若有侵权，请即时同我联系，我一定马上删除！！原创文章版权所有！如需转载，请注明出处： tekkamanninja.blog.chinaunix.net ，谢谢合作！！！ 拒绝一切广告性质的评论，一经发现立即举报并删除！

首页 | 博文目录 | 关于我



tekkamanninj

博客访问： 75917
博文数量： 263
博客积分： 15936
博客等级： 上将
技术积分： 13951
用户组： 普通用户
注册时间： 2007-03-27 11:22

[加关注](#)[短消息](#)

[论坛](#)[加好友](#)

个人简介
Fedora-ARM

文章分类

- 全部博文（263）
- Red Hat（2）

代码管理（6）

感悟（3）

Linux调试技术（2）

MaxWit（1）

Linux设备驱动程（41）

Android（20）

neo freerunner（2）

计算机硬件技术（9）

网络（WLAN or LA（8）

励志（7）

ARM汇编语言（1）

Linux操作系统的（15）

Linux内核研究（38）

ARM-Linux应用程（19）

建立根文件系统（4）

Linux内核移植（14）

Bootloader（45）

建立ARM-Linux交（7）

未分配的博文（19）

文章存档

2014年（1）

Linux设备驱动程序学习（8）-分配内存

2007-11-22 18:27:44

分类： LINUX

Linux设备驱动程序学习（8）-分配内存

内核为设备驱动提供了一个统一的内存管理接口，所以模块无需涉及分段和分页等问题。 我已经在第一个scull模块中使用了 kmalloc 和 kfree 来分配和释放内存空间。

kmalloc 函数内幕

kmalloc 是一个功能强大且高速(除非被阻塞)的工具，所分配到的内存在物理内存中连续且保持原有的数据（不清零）。原型：

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

size 参数

内核管理系统的物理内存，物理内存只能按页面进行分配。kmalloc 和典型的用户空间 malloc 在实际上有很大的差别，内核使用特殊的基于页的分配技术，以最佳的方式利用系统 RAM。Linux 处理内存分配的方法：创建一系列内存对象集合，每个集合内的内存块大小是固定。处理分配请求时，就直接在包含有足够大内存块的集合中传递一个整块给请求者。

必须注意的是：内核只能分配一些预定义的、固定大小的字节数组。kmalloc 能够处理的最小内存块是 32 或 64 字节（体系结构依赖），而内存块大小的上限随着体系和内核配置而变化。考虑到移植性，不应分配大于 128 KB的内存。若需多于几个 KB的内存块，最好使用其他方法。

flags 参数

内存分配最终总是调用 __get_free_pages 来进行实际的分配，这就是 GFP_ 前缀的由来。

所有标志都定义在 <linux/gfp.h>，有符号代表常常使用的标志组合。

主要的标志常被称为分配优先级，包括：

GFP_KERNEL

最常用的标志，意思是这个分配代表运行在内核空间的进程进行。内核正常分配内存。当空闲内存较少时，可能进入休眠来等待一个页面。当前进程休眠时，内核会采取适当的动作来获取空闲页。所以使用 GFP_KERNEL 来分配内存的函数必须是可重入，且不能在原子上下文运行。

GFP_ATOMIC

内核通常会为原子性的分配预留一些空闲页。当当前进程不能被置为睡眠时，应使用 GFP_ATOMIC，这样 kmalloc 甚至能够使用最后一个空闲页。如果连这最后一个空闲页也不存在，则分配返回失败。常用来从中断处理和进程上下文之外的其他代码中分配内存，从不睡眠。

GFP_USER

用来为用户空间分配内存页，可能睡眠。

GFP_HIGHUSER

类似 GFP_USER，如果有高端内存，就从高端内存分配。

GFP_NOIO

GFP_NOFS


功能类似 GFP_KERNEL，但是为内核分配内存的工作增加了限制。具有GFP_NOFS 的分配不允许执行任何文件系统调用，而 GFP_NOIO 禁止任何 I/O 初始化。它们主要用在文件系统和虚拟内存代码。那里允许分配休眠，但不应发生递归的文件系统调。

2013年（3）
2012年（61）
2011年（66）
2010年（27）
2009年（30）
2008年（23）
2007年（52）

我的朋友



小蜗牛快



cfm5538




jikaishi



shizhenc



pxy05215



李怀远



yan19900



wkm81018



xiousi

最近访客



apang199




appcount



zaichu




lhxzui



小蜗牛快



小尾巴鱼



erain_30



hushup



wilfred_

订阅

推荐博文

- linux 3.x的 通用时钟架构 ...
- SCN的相关解析
- Flash驱动学习
- 浅谈nagios之state type和 no...
- DB2（Linux 64位）安装教程...
- insert语句造成latch:library...
- 2014.06.13 网络公开课《让我...
- MySQL Slave异常关机的处理（...
- 巧用shell脚本分析数据库用户...
- 查询linux、HP-UX的cpu信息...

热词专题

- linux系统权限修复——学生误...
- Modbus协议使用
- linux
- busybox原理
- php环境搭建教程

Linux 内核把内存分为 3 个区段：可用于DMA的内存（位于一个特别的地址范围的内存，外设可以在这里进行 DMA 存取）、常规内存和高端内存（为了访问（相对）大量的内存而存在的一种机制）。目的是使每中计算机平台都必须知道如何将自己特定的内存范围归类到这三个区段中，而不是所有RAM都一样。

当要分配一个满足kmalloc要求的新页时，内核会建立一个内存区段的列表以供搜索。若指定了 __GFP_DMA，只有可用于DMA的内存区段被搜索；若没有指定特别的标志，常规和 可用于DMA的内存区段都被搜索；若 设置了 __GFP_HIGHMEM，所有的 3 个区段都被搜索（注意：kmalloc 不能分配高端内存）。

内存区段背后的机制在 mm/page_alloc.c 中实现，且区段的初始化时平台相关的，通常在 arch 目录树的 mm/init.c中。

有的标志用双下划线做前缀，他们可与上面标志“或”起来使用，以控制分配方式：

__GFP_DMA

要求分配可用于DMA的内存。

__GFP_HIGHMEM

分配的内存可以位于高端内存。

__GFP_COLD

通常，分配器试图返回“缓存热（cache warm）”页面（可在处理器缓存中找到的页面）。而这个标志请求一个尚未使用的“冷”页面。对于用作 DMA 读取的页面分配，可使用此标志。因为此时页面在处理器缓存中没多大帮助。

__GFP_NOWARN

当一个分配无法满足，阻止内核发出警告（使用 printk）。

__GFP_HIGH

高优先级请求，允许为紧急状况消耗被内核保留的最后一些内存页。

__GFP_REPEAT

__GFP_NOFAIL

__GFP_NORETRY

告诉分配器当满足一个分配有困难时，如何动作。__GFP_REPEAT 表示努力再尝试一次，仍然可能失败；__GFP_NOFAIL 告诉分配器尽最大努力来满足要求，始终不返回失败，不推荐使用；__GFP_NORETRY 告知分配器如果无法满足请求，立即返回。

后备高速缓存

内核为驱动程序常常需要反复分配许多相同大小内存块的情况，增加了一些特殊的内存池，称为后备高速缓存（lookaside cache）。设备驱动程序通常不会涉及后备高速缓存，但是也有例外：在 Linux 2.6 中 USB 和 SCSI 驱动。Linux 内核的高速缓存管理器有时称为“slab 分配器”，相关函数和类型在 <linux/slab.h> 中声明。slab 分配器实现的高速缓存具有 kmem_cache_t 类型。实现过程如下：

（1）创建一个新的后备高速缓存

```
kmem_cache_t *kmem_cache_create(const char *name, size_t size, size_t offset,
                                unsigned long flags,
                                void (*constructor)(void *, kmem_cache_t *, unsigned long flags),
                                void (*destructor)(void *, kmem_cache_t *, unsigned long flags));
/*创建一个可以容纳任意数目内存区域的、大小都相同的高速缓存对象*/
```

参数name： 一个指向 name 的指针，name和这个后备高速缓存相关联，功能是管理信息以便追踪问题；通常设置为被缓存的结构类型的名字，不能包含空格。

参数size： 每个内存区域的大小。

参数offset： 页内第一个对象的偏移量；用来确保被分配对象的特殊对齐, 0 表示缺省值。

参数flags： 控制分配方式的位掩码：

SLAB_NO_REAP 保护缓存在系统查找内存时不被削减，不推荐。

SLAB_HWCACHE_ALIGN 所有数据对象跟高速缓存行对齐，平台依赖，可能浪费内存。

SLAB_CACHE_DMA 每个数据对象在 DMA 内存区段分配。

其他标志详见 mm/slab.c。但是, 通常这些标志在只在开发系统中通过内核配置选项被全局性地设置。

参数constructor 和 destructor 是可选函数(不能只有destructor，而没有constructor)，用来初始化新分配的对象和在内存被作为整体释放给系统之前“清理”对象。

constructor 函数在分配一组对象的内存时被调用，由于内存可能持有几个对象，所以可能被多次调用。

同理，destructor不是立刻在一个对象被释放后调用，而可能在以后某个未知的时间内调用。根据它们是否被传递 SLAB_CTOR_ATOMIC 标志（CTOR 是 constructor 的缩写），控制是否允许休眠。由于当被调用者是constructor函数时，slab 分配器会传递 SLAB_CTOR_CONSTRUCTOR 标志。为了方便，它们可通过检测这个标志以使用同一函数。

(2) 通过调用 kmem_cache_alloc 从已创建的后备高速缓存中分配对象：

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
/*cache 参数是刚创建缓存，flags 是和kmallocc 的相同*/
```

(3) 使用 kmem_cache_free释放一个对象：

```
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

(4) 当驱动用完这个后备高速缓存（通常在当模块被卸载时），释放缓存：

```
int kmem_cache_destroy(kmem_cache_t *cache);
/*只在从这个缓存中分配的所有的对象都已延时才成功。因此，应检查 kmem_cache_destroy 的返回值：失败指示模块存在内存泄漏*/
```

使用后备高速缓存的一个好处是内核会统计后备高速缓存的使用，统计情况可从 /proc/slabinfo 获得。

内存池

为了确保在内存分配不允许失败情况下成功分配内存，内核提供了称为内存池（“mempool”）的抽象，它其实是某种后备高速缓存。它为了紧急情况下的使用，尽力一直保持空闲内存。所以使用时必须注意：mempool 会分配一些内存块，使其空闲而不真正使用，所以容易消耗大量内存。而且不要使用 mempool 处理可能失败的分配。应避免在驱动代码中使用 mempool。

内存池的类型为 mempool_t，在 <linux/mempool.h>，使用方法如下：

(1) 创建 mempool：

```
mempool_t *mempool_create(int min_nr,
                           mempool_alloc_t *alloc_fn,
                           mempool_free_t *free_fn,
                           void *pool_data);
/*min_nr 参数是内存池应当一直保留的最小数量的分配对象*/

/*实际的分配和释放对象由 alloc_fn 和 free_fn 处理, 原型:*/
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
typedef void (mempool_free_t)(void *element, void *pool_data);
/*给 mempool_create 最后的参数 *pool_data 被传递给 alloc_fn 和 free_fn */
```

你可编写特殊用途的函数来处理 mempool 的内存分配，但通常只需使用 slab 分配器为你处理这个任务：mempool_alloc_slab 和 mempool_free_slab的原型和上述内存池分配原型匹配，并使用 kmem_cache_alloc 和 kmem_cache_free 处理内存的分配和释放。

典型的设置内存池的代码如下：

```
cache = kmem_cache_create(. . .);
pool = mempool_create(MY_POOL_MINIMUM, mempool_alloc_slab, mempool_free_slab, cache);
```

(2) 创建内存池后，分配和释放对象：

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

在创建mempool时，分配函数将被调用多次来创建预先分配的对象。因此，对 mempool_alloc 的调用是试图用分配函数请求额外的对象，如果失败，则返回预先分配的对象(如果存在)。用 mempool_free 释放对象时，若预先分配的对象数目小于最小量，就将它保留在池中，否则将它返回给系统。

可用一下函数重定义mempool预先分配对象的数量：

```
int mempool_resize(mempool_t *pool, int new_min_nr, int gfp_mask);
/*若成功，内存池至少有 new_min_nr 个对象*/
```

(3) 若不再需要内存池，则返回给系统：

```
void mempool_destroy(mempool_t *pool);
/*在销毁 mempool 之前，必须返回所有分配的对象，否则会产生 oops*/
```

get_free_page 和相关函数

如果一个模块需要分配大块的内存，最好使用面向页的分配技术。

```

__get_free_page(unsigned int flags);
/*返回一个指向新页的指针，未清零该页*/

get_zeroed_page(unsigned int flags);
/*类似于__get_free_page，但用零填充该页*/

__get_free_pages(unsigned int flags, unsigned int order);
/*分配是若干(物理连续的)页面并返回指向该内存区域的第一个字节的指针，该内存区域未清零*/

/*参数flags 与 kmalloc 的用法相同；
参数order 是请求或释放的页数以 2 为底的对数。若其值过大(没有这么大的连续区可用)，则分配失败*/

```

get_order 函数可以用来从一个整数参数 size(必须是 2 的幂)中提取 order，函数也很简单：

```

/* Pure 2^n version of get_order */
static __inline__ __attribute_const__ int get_order(unsigned long size)
{
    int order;

    size = (size - 1) >> (PAGE_SHIFT - 1);
    order = -1;
    do {
        size >>= 1;
        order++;
    } while (size);
    return order;
}

```

使用方法举例在Linux设备驱动程序学习 (7) - 内核的数据类型 中有。

通过/proc/buddyinfo 可以知道系统中每个内存区段上的每个 order 下可获得的数据块数目。

当程序不需要页面时，它可用下列函数之一来释放它们。

```

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

```

它们的关系是：

```

#define __get_free_page(gfp_mask) \
    __get_free_pages((gfp_mask), 0)

```

若试图释放和你分配的数目不等的页面，会破坏内存映射关系，系统会出错。

注意：只要符合和 kmalloc 的相同规则，__get_free_pages 和其他的函数可以在任何时候调用。这些函数可能失败（特别当使用 GFP_ATOMIC 时），因此调用这些函数的程序必须提供分配失败的处理。

从用户的角度，可感觉到的区别主要是速度提高和更好的内存利用率(因为没有内部的内存碎片)。但主要优势实际不是速度，而是更有效的内存利用。__get_free_page 函数的最大优势是获得的页完全属于调用者，且理论上可以适当的设置页表将起合并成一个线性的区域。

alloc_pages 接口

struct page 是一个描述一个内存页的内部内核结构，定义在<linux/Mm_types.h>。

Linux 页分配器的核心是称为 alloc_pages_node 的函数：

```

struct page *alloc_pages_node(int nid, unsigned int flags,
    unsigned int order);

/*以下是这个函数的 2 个变体(是简单的宏)：*/
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);

/*他们的关系是：*/

```

```
#define alloc_pages(gfp_mask, order) \
    alloc_pages_node(numa_node_id(), gfp_mask, order)
#define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)
```

参数nid 是要分配内存的 NUMA 节点 ID,

参数flags 是 GFP_ 分配标志,

参数order 是分配内存的大小.

返回值是一个指向第一个(可能返回多个页)page结构的指针, 失败时返回NULL。

alloc_pages 通过在当前 NUMA 节点分配内存(它使用 numa_node_id 的返回值作为 nid 参数调用 alloc_pages_node)简化了alloc_pages_node调用。alloc_pages 省略了 order 参数而只分配单个页面。

释放分配的页:

```
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
void free_cold_page(struct page *page);
/*若知道某个页中的内容是否驻留在处理器高速缓存中, 可以使用 free_hot_page (对于驻留在缓存中的页) 或 free_cold_page(对于没有驻留在缓存中的页) 通知内核, 帮助分配器优化内存使用*/
```

vmalloc 和 ioremap

vmalloc 是一个基本的 Linux 内存分配机制, 它在虚拟内存空间分配一块连续的内存区, 尽管这些页在物理内存中不连续 (使用一个单独的 alloc_page 调用来获得每个页), 但内核认为它们地址是连续的。应当注意的是: vmalloc 在大部分情况下不推荐使用。因为在某些体系上留给 vmalloc 的地址空间相对小, 且效率不高。函数原型如下:

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

kmalloc 和 _get_free_pages 返回的内存地址也是虚拟地址, 其实际值仍需 MMU 处理才能转为物理地址。vmalloc和它们在使用硬件上没有不同, 不同是在内核如何执行分配任务上: kmalloc 和 _get_free_pages 使用的(虚拟)地址范围和物理内存是一对一映射的, 可能会偏移一个常量 PAGE_OFFSET 值, 无需修改页表。

而vmalloc 和 ioremap 使用的地址范围完全是虚拟的, 且每次分配都要通过适当地设置页表来建立(虚拟)内存区域。 vmalloc 可获得的地址在从 VMALLOC_START 到 VMLLOC_END 的范围中, 定义在 <asm/patable.h> 中。vmalloc 分配的地址只在处理器的 MMU 之上才有意义。当驱动需要真正的物理地址时, 就不能使用 vmalloc。 调用 vmalloc 的正确场合是分配一个大的、只存在于软件中的、用于缓存的内存区域时。注意: vamlloc 比 __get_free_pages 要更多开销, 因为它必须即获取内存又建立页表。因此, 调用 vmalloc 来分配仅仅一页是不值得的。vmalloc 的一个小的缺点在于它无法在原子上下文中使用。因为它内部使用 kmalloc(GFP_KERNEL) 来获取页表的存储空间, 因此可能休眠。

ioremap 也要建立新页表, 但它实际上不分配任何内存, 其返回值是一个特殊的虚拟地址可用来访问特定的物理地址区域。

保持了可移植性, 不应当像访问内存指针一样直接访问由 ioremap 返回的地址, 而应当始终使用 readb 和其他 I/O 函数。

ioremap 和 vmalloc 是面向页的(它们会修改页表), 重定位的或分配的空间都会被上调到最近的页边界。ioremap 通过将重映射的地址下调到页边界, 并返回第一个重映射页内的偏移量来模拟一个非对齐的映射。

per-CPU 的变量

per-CPU 变量是一个有趣的 2.6 内核特性, 定义在 <linux/percpu.h> 中。当创建一个per-CPU变量, 系统中每个处理器都会获得该变量的副本。其优点是对per-CPU变量的访问(几乎)不需要加锁, 因为每个处理器都使用自己的副本。per-CPU 变量也可存在于它们各自的处理器缓存中, 这就在频繁更新时带来了更好性能。

在编译时间创建一个per-CPU变量使用如下宏定义:


```
DEFINE_PER_CPU(type, name);  
/*若变量( name)是一个数组, 则必须包含类型的维数信息, 例如一个有 3 个整数的per-CPU 数组  
创建如下: */  
DEFINE_PER_CPU(int[3], my_percpu_array);
```

虽然操作per-CPU变量几乎不必使用锁定机制。但是必须记住 2.6 内核是可抢占的, 所以在修改一个per-CPU变量的临界区中可能被抢占。并且还要避免进程在对一个per-CPU变量访问时被移动到另一个处理器上运行。所以必须显式使用 `get_cpu_var` 宏来访问当前处理器的变量副本, 并在结束后调用 `put_cpu_var`。对 `get_cpu_var` 的调用返回一个当前处理器变量版本的 `lvalue`, 并且禁止抢占。又因为返回的是`lvalue`, 所以可被直接赋值或操作。例如:

```
get_cpu_var(sockets_in_use)++;  
put_cpu_var(sockets_in_use);
```

当要访问另一个处理器的变量副本时, 使用:

```
per_cpu(variable, int cpu_id);
```

当代码涉及到多处理器的per-CPU变量, 就必须实现一个加锁机制来保证访问安全。

动态分配per-CPU变量方法如下:

```
void *alloc_percpu(type);  
void *_alloc_percpu(size_t size, size_t align);/*需要一个特定对齐的情况下调用*/  
void free_percpu(void *per_cpu_var); /* 将per-CPU 变量返回给系统*/  
  
/*访问动态分配的per-CPU变量通过 per_cpu_ptr 来完成, 这个宏返回一个指向给定 cpu_id 版本的  
per_cpu_var变量的指针。若操作当前处理器版本的per-CPU变量, 必须保证不能被切换出那个处  
理器:*/  
per_cpu_ptr(void *per_cpu_var, int cpu_id);  
  
/*通常使用 get_cpu 来阻止在使用per-CPU变量时被抢占, 典型代码如下:*/  
  
int cpu;  
cpu = get_cpu()  
ptr = per_cpu_ptr(per_cpu_var, cpu);  
/* work with ptr */  
put_cpu();  
  
/*当使用编译时的per-CPU 变量, get_cpu_var 和 put_cpu_var 宏将处理这些细节。动态per-CPU  
变量需要更明确的保护*/
```

per-CPU变量可以导出给模块, 但必须使用一个特殊的宏版本:

```
EXPORT_PER_CPU_SYMBOL(per_cpu_var);  
EXPORT_PER_CPU_SYMBOL_GPL(per_cpu_var);  
  
/*要在模块中访问这样一个变量, 声明如下:*/  
DECLARE_PER_CPU(type, name);
```

注意: 在某些体系架构上, per-CPU变量的使用是受地址空间有限的。若在代码中创建per-CPU变量, 应当尽量保持变量较小。

获得大的缓冲区

大量连续内存缓冲的分配是容易失败的。到目前止执行大 I/O 操作的最好方法是通过离散/聚集操作。

在引导时获得专用缓冲区

若真的需要大块连续的内存作缓冲区, 最好的方法是在引导时来请求内存来分配。在引导时分配是获得大量连续内存页(避开 `__get_free_pages` 对缓冲大小和固定颗粒双重限制)的唯一方法。一个模块无法在

引导时分配内存，只有直接连接到内核的驱动才可以。而且这对普通用户不是一个灵活的选择，因为这个机制只对连接到内核映像中的代码才可用。要安装或替换使用这种分配方法的设备驱动，只能通过重新编译内核并且重启计算机。

当内核被引导，它可以访问系统中所有可用物理内存，接着通过调用子系统的初始化函数，允许初始化代码通过减少留给常规系统操作使用的 RAM 数量来分配私有内存缓冲给自己。

在引导时获得专用缓冲区要通过调用下面函数进行：

```
#include <linux/bootmem.h>
/*分配不在页面边界上对齐的内存区*/
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size); /*分配非高端内存。希望分配到用于DMA操作的内存可能需要，因为高端内存不总是支持DMA*/

/*分配整个页*/
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size); /*分配非高端内存*/

/*很少在启动时释放分配的内存，但肯定不能在之后取回它。注意：以这个方式释放的部分页不返回给系统*/
void free_bootmem(unsigned long addr, unsigned long size);
```

更多细节请看内核源码中 Documentation/kbuild 下的文件。

ARM9开发板实验

这几个实验我都没有用LDD3原配的代码，而是用以前的ioctl_and_lseek的代码改的。

(1) scullc 模块试验

模块源码：scullc

测试代码：scullc_test

(2) sculpl 模块试验

模块源码：sculpl

测试代码：sculpl_test

(3) scullv 模块试验

模块源码：scullv

实验现象：

```
[Tekkaman2440@SBC2440V4]#insmod /lib/modules/scullc.ko
[Tekkaman2440@SBC2440V4]#cat /proc/devices
Character devices:
 1 mem
 2 pty
 3 tty
 4 /dev/vc/0
 4 tty
 4 ttyS
 5 /dev/tty
 5 /dev/console
 5 /dev/ptmx
 7 vcs
10 misc
13 input
14 sound
81 video4linux
89 i2c
```

```

90 mtd
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
204 s3c2410_serial
252 scullc
253 usb_endpoint
254 rtc

Block devices:
 1 ramdisk
256 rfd
 7 loop
31 mtdblock
93 nftl
96 inftl
179 mmc
[Tekkaman2440@SBC2440V4]#mknod -m 666 scullc c 252 0
[Tekkaman2440@SBC2440V4]#ls -l /tmp/test (注: test是普通的2070B的文本文件)
-rw-r--r-- 1 root root 2070 Nov 24 2007 /

[Tekkaman2440@SBC2440V4]#cat /tmp/test > /dev/scullc
[Tekkaman2440@SBC2440V4]#cat /proc/slabinfo
slabinfo - version: 2.1 (statistics)
# name <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables
<limit> <batchcount> <sharedfactor> : slabdata <active_slabs> <num_slabs>
<sharedavail> : globalstat <listallocs> <maxobjs> <grown> <reaped> <error>
<maxfreeable> <nodeallocs> <remote frees> <alienoverflow> : cpustat <allochit>
<allocmiss> <freehit> <freemiss>
scullc 1 1 4020 1 1 : tunables 24 12 0 : slabdata 1 1 0 : globalstat 1 1 1 0 0 0 0 0
: cpustat 0 1 0 0
.....
[Tekkaman2440@SBC2440V4]#cat /proc/scullcseq

Device 0: qset 1000, q 4000, sz 2070
  item at c3edca00, qset at c3efe000
    0: c3f56028
[Tekkaman2440@SBC2440V4]#/tmp/scullc_test
open scull !
scull_quantum=10 scull_qset=4
close scull !

[Tekkaman2440@SBC2440V4]#cat /tmp/test > /dev/scullc
[Tekkaman2440@SBC2440V4]#cat /proc/slabinfo
slabinfo - version: 2.1 (statistics)
# name <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables
<limit> <batchcount> <sharedfactor> : slabdata <active_slabs> <num_slabs>
<sharedavail> : globalstat <listallocs> <maxobjs> <grown> <reaped> <error>
<maxfreeable> <nodeallocs> <remote frees> <alienoverflow> : cpustat <allochit>
<allocmiss> <freehit> <freemiss>
scullc 207 207 4020 1 1 : tunables 24 12 0 : slabdata 207 207 0 : globalstat 208 207
208 1 0 0 0 0 0 : cpustat 0 208 1 0
.....
[Tekkaman2440@SBC2440V4]#cat /proc/scullcmem

Device 0: qset 4, q 10, sz 2070
  item at c3edc930, qset at c3edc8c8

```



```
item at c3edc894, qset at c3edc860
item at c3edc82c, qset at c3edc7f8
item at c3edc7c4, qset at c3edc790
item at c3edc75c, qset at c3edc178
item at c3edc624, qset at c3edc4b8
item at c3edc9cc, qset at c3edc998
item at c3edc4ec, qset at c3edc964
item at c3edc484, qset at c3edccd8
item at c3edcca4, qset at c3edcc70
item at c3edcc3c, qset at c3edcc08
item at c3edcbd4, qset at c3edcba0
item at c3edcb6c, qset at c3edcb38
item at c3edcb04, qset at c3edcad0
item at c3edca9c, qset at c3edca68
item at c3edca34, qset at c3edca00
item at c3edc8fc, qset at c3edcfb0
item at c3edcf7c, qset at c3edcf48
item at c3edcf14, qset at c3edcee0
item at c3edceac, qset at c3edce78
item at c3edce44, qset at c3edce10
item at c3edcddc, qset at c3edcda8
item at c3edcd74, qset at c3edcd40
item at c3edcd0c, qset at c388a450
item at c388a41c, qset at c388a3e8
item at c388a3b4, qset at c388a380
item at c388a34c, qset at c388a318
item at c388a2e4, qset at c388a2b0
item at c388a27c, qset at c388a248
item at c388a214, qset at c388a1e0
item at c388a1ac, qset at c388a178
item at c388a144, qset at c388a790
item at c388a75c, qset at c388a728
item at c388a6f4, qset at c388a6c0
item at c388a68c, qset at c388a658
item at c388a624, qset at c388a5f0
item at c388a5bc, qset at c388a588
item at c388a554, qset at c388a520
item at c388a4ec, qset at c388a4b8
item at c388a484, qset at c388aad0
item at c388aa9c, qset at c388aa68
item at c388aa34, qset at c388aa00
item at c388a9cc, qset at c388a998
item at c388a964, qset at c388a930
item at c388a8fc, qset at c388a8c8
item at c388a894, qset at c388a860
item at c388a82c, qset at c388a7f8
item at c388a7c4, qset at c388ae10
item at c388addc, qset at c388ada8
item at c388ad74, qset at c388ad40
item at c388ad0c, qset at c388acd8
item at c388aca4, qset at c388ac70
0: c38fb028
1: c38fc028
2: c38fd028
[Tekkaman2440@SBC2440V4]#insmod /lib/modules/scullp.ko
[Tekkaman2440@SBC2440V4]#cat /proc/devices
Character devices:
1 mem
2 pty
3 tty
```

```
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
7 vcs
10 misc
13 input
14 sound
81 video4linux
89 i2c
90 mtd
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
204 s3c2410_serial
251 scullp
252 sculld
253 usb_endpoint
254 rtc

Block devices:
1 ramdisk
256 rfd
7 loop
31 mtdblock
93 nftl
96 inftl
179 mmc
[Tekkaman2440@SBC2440V4]#mknod -m 666 /dev/scullp c 251 0
[Tekkaman2440@SBC2440V4]#cat /tmp/test > /dev/scullp
[Tekkaman2440@SBC2440V4]#cat /proc/scullpseq

Device 0: qset 1000, q 4000, sz 2070
item at c3edc964, qset at c3eff000
0: c3f21000
[Tekkaman2440@SBC2440V4]#/tmp/scullp_test
open scull !
scull_quantum=1000 scull_qset=2
close scull !

[Tekkaman2440@SBC2440V4]#cat /tmp/test > /dev/scullp
[Tekkaman2440@SBC2440V4]#cat /proc/scullpseq

Device 0: qset 2, q 1000, sz 2070
item at c3edcc70, qset at c3edcca4
item at c3edc484, qset at c3edcc3c
0: c383f000
[Tekkaman2440@SBC2440V4]#insmod /lib/modules/scullv.ko
[Tekkaman2440@SBC2440V4]#cat /proc/devices

Character devices:
1 mem
2 pty
3 tty
4 /dev/vc/0
4 tty
```

```
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
7 vcs
10 misc
13 input
14 sound
81 video4linux
89 i2c
90 mtd
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
204 s3c2410_serial
250 scullv
251 sculpl
252 sculld
253 usb_endpoint
254 rtc

Block devices:
 1 ramdisk
256 rfd
 7 loop
31 mtdblock
93 nftl
96 inftl
179 mmc
[Tekkaman2440@SBC2440V4]#mknod -m 666 /dev/scullv c 250 0
[Tekkaman2440@SBC2440V4]#cat /tmp/test > /dev/scullv
[Tekkaman2440@SBC2440V4]#cat /proc/scullvseq

Device 0: qset 1000, q 4000, sz 2070
  item at c3edc7c4, qset at c389d000
    0: c485e000
[Tekkaman2440@SBC2440V4]#cat /proc/scullpseq

Device 0: qset 2, q 1000, sz 2070
  item at c3edcc70, qset at c3edcca4
  item at c3edc484, qset at c3edcc3c
    0: c383f000
```

从上面的数据可以看出：在s3c2440（ARM9）体系上，`vmalloc`返回的虚拟地址就在用于映射物理内存的地址上。

阅读(7285) | 评论(3) | 转发(31) |

上一篇：从PC总线到ARM的内部总线

下一篇：Linux设备驱动程序学习 (10) -时间、延迟及延缓操作

0

相关热门文章

云存储能否成为数据的保护神...

linux 常见服务端口

移植 ushare 到开发板

恢复他人删除聊天记录...	【ROOTFS搭建】busybox的httpd...	系统提供的库函数存在内存泄漏...
杀毒后清空的回收站如何恢复文...	xmanager 2.0 for linux配置	linux虚拟机 求教
Linux设备驱动程序学习...	什么是shell	初学UNIX环境高级编程的，关于...
Linux设备驱动程序学习（1）-...	linux socket的bug??	chinaunix博客什么时候可以设...

给主人留下些什么吧！~~



ycz9999 2012-07-24 13:57:09

tekkamanninja: 仁兄，说的没错。vmalloc() 返回的地址是还没有作为内核逻辑地址（虚拟地址）的虚拟地址空间中的地址，对于这些的理解请参考下：http://blog.chinaunix.net/uid-....
感谢前辈解答我的疑惑，我一直在跟着您的blog学习，您说的那篇文章，我也会好好学习的，希望以后前辈可以多多指点。

回复 | 举报



tekkamanninja 2012-07-24 08:52:05

ycz9999: 前辈，您好：
在这一节的学习中，我有一个疑问：
就是在您博客的最后，“从上面的数据可以看出：在s3c2440（ARM9）体系上，vmalloc返回的虚拟地址就在用于映射物理内存的地址上。”
仁兄，说的没错。vmalloc() 返回的地址是还没有作为内核逻辑地址（虚拟地址）的虚拟地址空间中的地址，对于这些的理解请参考下：http://blog.chinaunix.net/uid-20543672-id-2901339.html
一般这些地址对应的是非连续的物理内存地址，由MMU和页表负责转换。
这篇文章是我5年前写的，当时初学Linux，对这些的理解非常浅，所以难免有些问题，请见谅。

回复 | 举报



ycz9999 2012-07-24 08:16:19

前辈，您好：
在这一节的学习中，我有一个疑问：
就是在您博客的最后，“从上面的数据可以看出：在s3c2440（ARM9）体系上，vmalloc返回的虚拟地址就在用于映射物理内存的地址上。”

我的理解是：
vmalloc() 返回的虚拟地址应该是位于非连续内存区内，这段内存区域的起始地址是VMALLOC_START，末尾地址是VMALLOC_END，而不应该是您说的“用于映射物理内存的地址上”。我使用的实验开发板是mini2440(64M SDRAM)，通过计算可知VMALLOC_START应该是在0xc480 0000处(保留计算)，而我的实验结果也验证了这个数据，使用vmalloc分配的量子是位于这个地址以上的内存区域，即非连续内存区。
不知道我的理解对不对？还请前辈解惑。

回复 | 举报

评论热议

请登录后再评论。
[登录](#) [注册](#)