

Linux/Android开发记录

学习、记录、分享Linux/Android开发技术

目录视图

摘要视图

RSS 订阅

个人资料



liuhaoyutz

访问: 80625次

积分: 1673分

排名: 第7877名

原创: 83篇

转载: 0篇

译文: 0篇

评论: 59条

博客声明

本博客文章均为原创，欢迎转载交流。转载请注明出处，禁止用于商业目的。

博客专栏

Android应用开发学习笔记

文章: 30篇

阅读: 17067

LDD3源码分析

文章: 17篇

阅读: 29970

文章分类

LDD3源码分析 (18)

ADC驱动 (1)

触摸屏驱动 (1)

LCD驱动 (1)

Linux设备模型 (8)

USB驱动 (0)

Android架构分析 (12)

Cocos2d-x (1)

C陷阱与缺陷 (3)

Android应用开发 (30)

Linux设备驱动程序架构分析 (8)

有奖征资源，博文分享有内涵

5月推荐博文汇总

大数据读书汇--获奖名单公布

2014 CSDN博文大赛

LDD3源码分析之内存映射

分类: LDD3源码分析

2012-04-12 09:45

2336人阅读

评论(5)

收藏

举报

struct

file

tree

module

permissions

list

作者: 刘昊昱

博客: <http://blog.csdn.net/liuhaoyutz>

编译环境: Ubuntu 10.10

内核版本: 2.6.32-38-generic-pae

LDD3源码路径: examples/simple/

本文分析LDD3第十五章介绍的内存映射模块simple。

一、simple模块编译

在2.6.32-38-generic-pae内核下编译simple模块时，会遇到一些问题，下面列出遇到的问题及解决办法。

执行make编译simple模块，会出现如下错误：



修改Makefile文件，把CFLAGS改为EXTRA_CFLAGS即可解决这个问题。再次编译，出现如下错误：

<http://blog.csdn.net/liuhaoyutz/article/details/7452289>

1/10

最新评论

- LDD3源码分析之内存映射
wzw88486969:
@jhlhonnng:unsigned long offset
= vma->vm_pgoff <v...
- Linux设备驱动程序架构分析之I2
teamos:看了你的i2c的几篇文
章，真是受益匪浅，虽然让自己
写还是ie不出来。非常感谢
- LDD3源码分析之块设备驱动程序
elecetan2011: 感谢楼主的精彩讲
解，受益匪浅啊！
- LDD3源码分析之slab高速缓存
donghuwuwei: 省去不少修改
的时间，真是太好了
- LDD3源码分析之时间与延迟操作
donghuwuwei: jit.c代码需要加上
一个头文件。
- LDD3源码分析之slab高速缓存
捧灰: 今天学到这里了，可是为什
么我没有修改源码一遍就通过了
额。。。内核版本是2.6.18-
53.el5-x...
- LDD3源码分析之字符设备驱动程序
捧灰: 参照楼主的博客在自学~谢
谢楼主！
- LDD3源码分析之调试技术
fantasyhujian: 分析的很清楚，
赞一个！
- LDD3源码分析之字符设备驱动程
fantasyhujian: 有时间再好好读
读，真的分析的不错！
- LDD3源码分析之hello.c与Makef
fantasyhujian: 写的很详细，对
初学者很有帮助！！

阅读排行

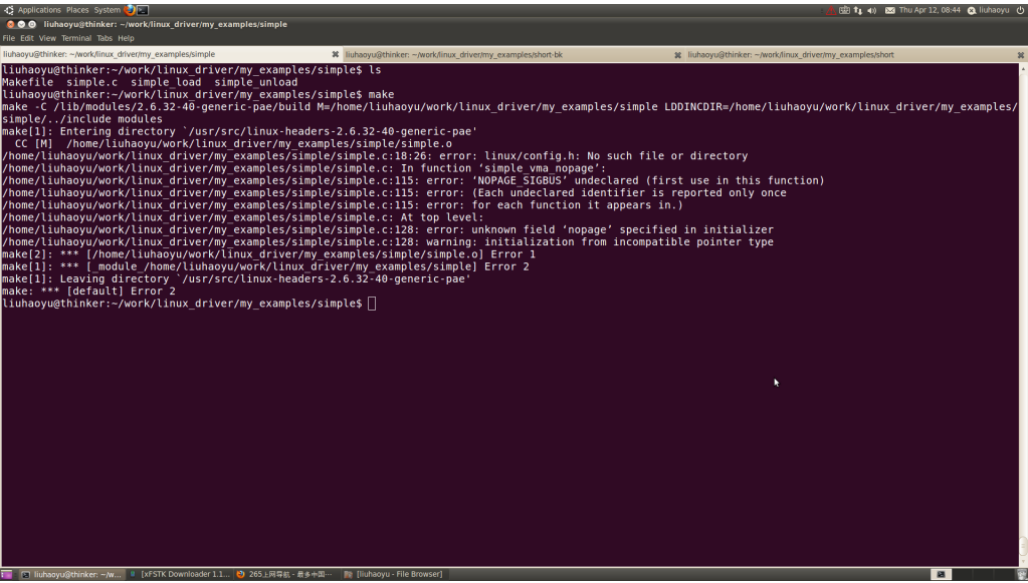
- LDD3源码分析之字符设: (3143)
- LDD3源码分析之hello.c: (2701)
- S3C2410驱动分析之LCI (2527)
- Linux设备模型分析之kse (2435)
- LDD3源码分析之内存映! (2336)
- LDD3源码分析之与硬件! (2333)
- Android架构分析之Andrc (2093)
- LDD3源码分析之时间与; (1987)
- LDD3源码分析之poll分析 (1972)
- S3C2410驱动分析之AD (1948)

评论排行

- LDD3源码分析之字符设: (12)
- S3C2410驱动分析之触摸 (7)
- LDD3源码分析之内存映! (5)
- LDD3源码分析之hello.c: (4)
- Linux设备模型分析之kot (4)
- LDD3源码分析之slab高; (4)
- S3C2410驱动分析之LCI (3)
- LDD3源码分析之阻塞型! (3)
- LDD3源码分析之时间与; (3)
- LDD3源码分析之poll分析 (2)

文章存档

- 2014年06月 (1)
- 2014年05月 (4)
- 2014年04月 (1)



修改simple.c把第18行#include <linux/config.h>屏蔽掉，即可解决第一个错误，再次编译，出现如下错误：



再次编译，出现如下错误：



[2014年01月](#) (1)[2013年12月](#) (6)[展开](#)

文章搜索

推荐文章

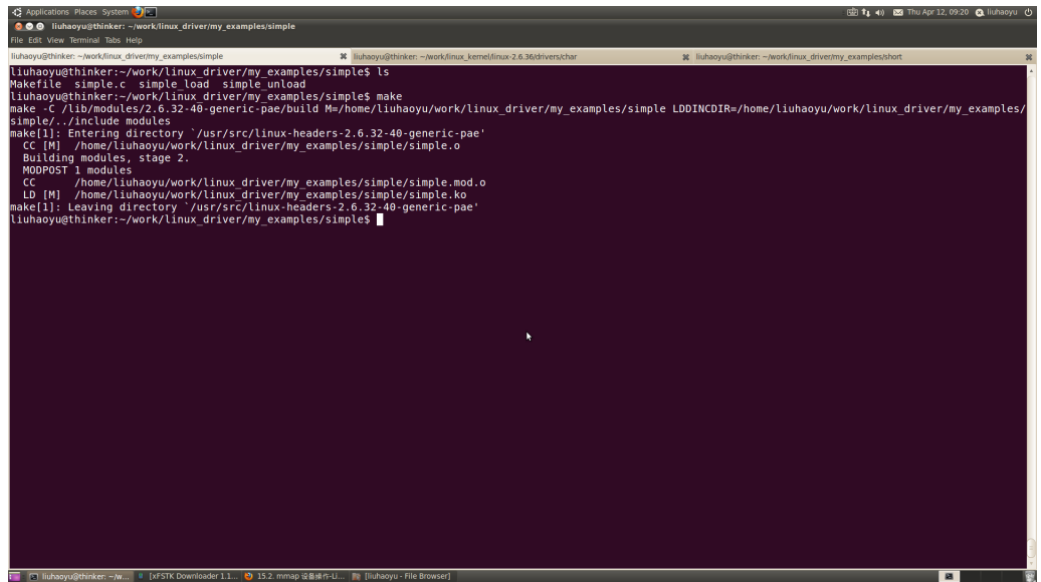
这是因为在新的内核中vm_operations_struct结构体的nopage函数已经被fault函数代替，所以把128行改为：

```
[cpp]
01. 128 .fault = simple_vma_nopage,
```

同时，按照fault的函数原型，重新改写simple_vma_nopage函数如下：

```
[cpp]
01. 102int simple_vma_nopage(struct vm_area_struct *vma,
02. 103                      struct vm_fault *vmf)
03. 104{
04. 105     struct page *pageptr;
05. 106     unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
06. 107     unsigned long physaddr = (unsigned long)vmf->virtual_address - vma-
>vm_start + offset;
07. 108     unsigned long pageframe = physaddr >> PAGE_SHIFT;
08. 109
09. 110// Eventually remove these printk
10. 111     printk (KERN_NOTICE "---- Nopage, off %lx phys %lx\n", offset, physaddr);
11. 112     printk (KERN_NOTICE "VA is %p\n", __va (physaddr));
12. 113     printk (KERN_NOTICE "Page at %p\n", virt_to_page (__va (physaddr)));
13. 114     if (!pfn_valid(pageframe))
14. 115         return 0;
15. 116     pageptr = pfn_to_page(pageframe);
16. 117     printk (KERN_NOTICE "page->index = %ld mapping %p\n", pageptr->index, pageptr-
>mapping);
17. 118     printk (KERN_NOTICE "Page frame %ld\n", pageframe);
18. 119     get_page(pageptr);
19. 120// if (type)
20. 121//     *type = VM_FAULT_MINOR;
21. 122     return VM_FAULT_NOPAGE;
22. 123}
```

再次编译，模块编译成功，如下图所示：



二、simple模块分析

首先来看simple模块初始化函数simple_init:

```
[cpp]
01. 197/*
02. 198 * Module housekeeping.
03. 199 */
04. 200static int simple_init(void)
05. 201{
06. 202     int result;
07. 203     dev_t dev = MKDEV(simple_major, 0);
08. 204
09. 205     /* Figure out our device number. */
10. 206     if (simple_major)
11. 207         result = register_chrdev_region(dev, 2, "simple");
12. 208     else {
13. 209         result = alloc_chrdev_region(&dev, 0, 2, "simple");
14. 210         simple_major = MAJOR(dev);
15. 211     }
16. 212     if (result < 0) {
17. 213         printk(KERN_WARNING "simple: unable to get major %d\n", simple_major);
18. 214         return result;
19. 215     }
20. 216     if (simple_major == 0)
21. 217         simple_major = result;
22. 218
23. 219     /* Now set up two cdevs. */
24. 220     simple_setup_cdev(SimpleDevs, 0, &simple_remap_ops);
25. 221     simple_setup_cdev(SimpleDevs + 1, 1, &simple_nopage_ops);
26. 222     return 0;
27. 223}
```

203 - 217行, 分配设备编号。

220行, 创建字符设备, 文件操作函数集是simple_remap_ops, 使用remap_pfn_range映射内存。

221行, 创建字符设备, 文件操作函数集是simple_nopage_ops, 使用nopage映射内存。

simple_setup_cdev函数定义如下:

```
[cpp]
01. 145/*
02. 146 * Set up the cdev structure for a device.
03. 147 */
04. 148static void simple_setup_cdev(struct cdev *dev, int minor,
05. 149                             struct file_operations *fops)
06. 150{
07. 151     int err, devno = MKDEV(simple_major, minor);
```

```

08. 152
09. 153     cdev_init(dev, fops);
10. 154     dev->owner = THIS_MODULE;
11. 155     dev->ops = fops;
12. 156     err = cdev_add (dev, devno, 1);
13. 157     /* Fail gracefully if need be */
14. 158     if (err)
15. 159         printk (KERN_NOTICE "Error %d adding simple%d", err, minor);
16. 160}

```

simple_setup_cdev函数关联字符设备文件操作函数集，并向内核注册字符设备。

下面先来看simple_remap_ops文件操作函数集：

```

[cpp]
01. 166/* Device 0 uses remap_pfn_range */
02. 167static struct file_operations simple_remap_ops = {
03. 168     .owner    = THIS_MODULE,
04. 169     .open     = simple_open,
05. 170     .release  = simple_release,
06. 171     .mmap     = simple_remap_mmap,
07. 172};

```

simple_open和simple_release函数的实现都是简单返回0。

```

[cpp]
01. 43static int simple_open (struct inode *inode, struct file *filp)
02. 44{
03. 45     return 0;
04. 46}
05.
06. 52static int simple_release(struct inode *inode, struct file *filp)
07. 53{
08. 54     return 0;
09. 55}

```

simple_remap_mmap函数的实现如下：

```

[cpp]
01. 85static int simple_remap_mmap(struct file *filp, struct vm_area_struct *vma)
02. 86{
03. 87     if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff,
04. 88         vma->vm_end - vma->vm_start,
05. 89         vma->vm_page_prot))
06. 90         return -EAGAIN;
07. 91
08. 92     vma->vm_ops = &simple_remap_vm_ops;
09. 93     simple_vma_open(vma);
10. 94     return 0;
11. 95}

```

这里我们要介绍一下，当用户空间调用mmap执行内存映射时，file_operations结构的mmap函数被调用，其函数原型是：

```

[cpp]
01. int (*mmap)(struct file *filp, struct vm_area_struct *vma);

```

其中vma包含了用于访问设备的虚拟地址信息。vma中vm_area_struct结构体类型，该结构体用于描述一个虚拟内存区，在2.6.10上，其定义如下：

```

[cpp]
01. 55/*
02. 56 * This struct defines a memory VMM memory area. There is one of these
03. 57 * per VM-area/task. A VM area is any part of the process virtual memory

```

```

04. 58 * space that has a special rule for the page-fault handlers (ie a shared
05. 59 * library, the executable area etc).
06. 60 */
07. 61struct vm_area_struct {
08. 62     struct mm_struct * vm_mm;    /* The address space we belong to. */
09. 63     unsigned long vm_start;      /* Our start address within vm_mm. */
10. 64     unsigned long vm_end;       /* The first byte after our end address
11. 65                                within vm_mm. */
12. 66
13. 67     /* linked list of VM areas per task, sorted by address */
14. 68     struct vm_area_struct *vm_next;
15. 69
16. 70     pgprot_t vm_page_prot;      /* Access permissions of this VMA. */
17. 71     unsigned long vm_flags;     /* Flags, listed below. */
18. 72
19. 73     struct rb_node vm_rb;
20. 74
21. 75     /*
22. 76      * For areas with an address space and backing store,
23. 77      * linkage into the address_space->i_mmap prio tree, or
24. 78      * linkage to the list of like vmas hanging off its node, or
25. 79      * linkage of vma in the address_space->i_mmap_nonlinear list.
26. 80      */
27. 81     union {
28. 82         struct {
29. 83             struct list_head list;
30. 84             void *parent; /* aligns with prio_tree_node parent */
31. 85             struct vm_area_struct *head;
32. 86         } vm_set;
33. 87
34. 88         struct prio_tree_node prio_tree_node;
35. 89     } shared;
36. 90
37. 91     /*
38. 92      * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
39. 93      * list, after a COW of one of the file pages. A MAP_SHARED vma
40. 94      * can only be in the i_mmap tree. An anonymous MAP_PRIVATE, stack
41. 95      * or brk vma (with NULL file) can only be in an anon_vma list.
42. 96      */
43. 97     struct list_head anon_vma_node; /* Serialized by anon_vma->lock */
44. 98     struct anon_vma *anon_vma; /* Serialized by page_table_lock */
45. 99
46. 100    /* Function pointers to deal with this struct. */
47. 101    struct vm_operations_struct * vm_ops;
48. 102
49. 103    /* Information about our backing store: */
50. 104    unsigned long vm_pgoff; /* Offset (within vm_file) in PAGE_SIZE
51. 105                           units, *not* PAGE_CACHE_SIZE */
52. 106    struct file * vm_file; /* File we map to (can be NULL). */
53. 107    void * vm_private_data; /* was vm_pte (shared mem) */
54. 108
55. 109#ifdef CONFIG_NUMA
56. 110    struct mempolicy *vm_policy; /* NUMA policy for the VMA */
57. 111#endif
58. 112};

```

vm_area_struct结构体描述的虚拟内存区介于vm_start和vm_end之间，vm_ops成员指向这个VMA的操作函数集，其类型为vm_operations_struct结构体，定义如下：

```

[cpp]
01. 170/*
02. 171 * These are the virtual MM functions - opening of an area, closing and
03. 172 * unmapping it (needed to keep files on disk up-to-date etc), pointer
04. 173 * to the functions called when a no-page or a wp-page exception occurs.
05. 174 */
06. 175struct vm_operations_struct {
07. 176     void (*open)(struct vm_area_struct * area);
08. 177     void (*close)(struct vm_area_struct * area);
09. 178     struct page * (*nopage)
10. 179     (struct vm_area_struct * area, unsigned long address, int *type);
11. 180     int (*populate)
12. 181     (struct vm_area_struct * area, unsigned long address, unsigned long len, pgprot_t prot, unsigned long pgoff, int nonblock);
13. 182     int (*set_policy)(struct vm_area_struct *vma, struct mempolicy *new);
14. 183     struct mempolicy * (*get_policy)(struct vm_area_struct *vma,

```

```

14.     183                unsigned long addr);
15.     184#endif
16.     185};

```

当用户调用mmap系统调用时, 内核会进行如下处理:

1. 在进程的虚拟空间查找一块VMA.
2. 将这块VMA进行映射.
3. 如果设备驱动程序中定义了mmap函数, 则调用它.
4. 将这个VMA插入到进程的VMA链表中.

内存映射工作大部分由内核完成, 驱动程序中的mmap函数只需要为该地址范围建立合适的页表, 并将vma->vm_ops替换为一系列的新操作就可以了。有两种建立页表的方法, 一是使用remap_pfn_range函数一次全部建立, 或者通过nopage方法每次建立一个页表。

simple_remap_mmap函数使用remap_pfn_range函数一次建立全部页表, remap_pfn_range函数原型如下:

```

[cpp]
01. int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_addr, unsigned long pfn,
    unsigned long size, pgprot_t prot);

```

vma代表虚拟内存区域。

virt_addr代表要建立页表的用户虚拟地址的起始地址, remap_pfn_range函数为处于virt_addr和virt_addr+size之间的虚拟地址建立页表。

pfn是与物理内存起始地址对应的页帧号, 虚拟内存将要被映射到该物理内存上。页帧号只是将物理地址右移PAGE_SHIFT位。在大多数情况下, VMA结构中的vm_pgoff赋值给pfn即可。remap_pfn_range函数建立页表, 对应的物理地址是pfn<<PAGE_SHIFT到pfn<<(PAGE_SHIFT)+size。

size代表虚拟内存区域大小。

port是VMA要求的protection属性, 驱动程序只要使用vma->vm_page_prot中的值即可。

在simple_remap_mmap函数中,

87 - 90行, 调用remap_pfn_range函数建立页表:

```

[cpp]
01. remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff,
02.                vma->vm_end - vma->vm_start,
03.                vma->vm_page_prot)

```

可对比上面对remap_pfn_range函数的参数的解释来理解。

92行, vma->vm_ops是struct vm_operations_struct类型变量, 代表内核操作虚拟内存区的函数集, 这里赋值为simple_remap_vm_ops, 其定义如下:

```

[cpp]
01. 80static struct vm_operations_struct simple_remap_vm_ops = {
02.     81     .open = simple_vma_open,
03.     82     .close = simple_vma_close,
04.     83};

```

这里仅仅实现了open和close函数, 其它函数由内核提供。当进程打开或关闭VMA时, 就会调用这两个函数, 当fork进程或者创建一个新的对VMA引用时, 也会调用open函数。实际的打开和关闭工作由内核完成, 这里实现的open和close函数不必重复内核所做的工作, 只要根据驱动程序的需要处理其他必要的事情。对于simple这样的简单驱动程序

序，simple_vma_open和simple_vma_close函数仅仅是打印相关信息：

```
[cpp]
01. 63void simple_vma_open(struct vm_area_struct *vma)
02. 64{
03. 65     printk(KERN_NOTICE "Simple VMA open, virt %lx, phys %lx\n",
04. 66                 vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
05. 67}
06. 68
07. 69void simple_vma_close(struct vm_area_struct *vma)
08. 70{
09. 71     printk(KERN_NOTICE "Simple VMA close.\n");
10. 72}
```

回到simple_remap_mmap函数：

93行，显式调用simple_vma_open(vma)，这里要注意，必须显示调用该函数，因为open函数还没有注册到系统。

至此，Device 0的相关代码就分析完了。

除了remap_pfn_range函数外，在驱动程序中实现nopage函数通常可以为设备提供更加灵活的内存映射途径。当访问的页面不在内存，即发生缺页中断时，nopage就会被调用。这是因为，当发生缺页中断时，系统会经过如下处理过程：

1. 找到缺页的虚拟地址所在的VMA。
2. 如果必要，分配中间页目录表和页表。
3. 如果页表项对应的物理页面不存在，则调用nopage函数，它返回物理页面的页描述符。
4. 将物理页面的地址填充到页表中。

下面我们看Device 1的相关代码，其文件操作函数集如下：

```
[cpp]
01. 174/* Device 1 uses nopage */
02. 175static struct file_operations simple_nopage_ops = {
03. 176     .owner    = THIS_MODULE,
04. 177     .open     = simple_open,
05. 178     .release  = simple_release,
06. 179     .mmap     = simple_nopage_mmap,
07. 180};
```

与Device 0相比，只有mmap的实现不一样，我们看simple_nopage_mmap：

```
[cpp]
01. 131static int simple_nopage_mmap(struct file *filp, struct vm_area_struct *vma)
02. 132{
03. 133     unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
04. 134
05. 135     if (offset >= __pa(high_memory) || (filp->f_flags & O_SYNC))
06. 136         vma->vm_flags |= VM_IO;
07. 137     vma->vm_flags |= VM_RESERVED;
08. 138
09. 139     vma->vm_ops = &simple_nopage_vm_ops;
10. 140     simple_vma_open(vma);
11. 141     return 0;
12. 142}
```

135 - 137行，设置vma->vm_flags标志。

139行，指定vma->vm_ops为simple_nopage_vm_ops。

140行，显式调用simple_vma_open函数。

simple_nopage_vm_ops结构定义如下：

[cpp]

```
01. 125static struct vm_operations_struct simple_nopage_vm_ops = {
02. 126     .open =     simple_vma_open,
03. 127     .close =    simple_vma_close,
04. 128     .nopage =    simple_vma_nopage,
05. 129};
```

这个结构体中，需要分析的是simple_vma_nopage函数：

[cpp]

```
01. 102struct page *simple_vma_nopage(struct vm_area_struct *vma,
02. 103                             unsigned long address, int *type)
03. 104{
04. 105     struct page *pageptr;
05. 106     unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
06. 107     unsigned long physaddr = address - vma->vm_start + offset;
07. 108     unsigned long pageframe = physaddr >> PAGE_SHIFT;
08. 109
09. 110// Eventually remove these prints
10. 111     printk (KERN_NOTICE "---- Nopage, off %lx phys %lx\n", offset, physaddr);
11. 112     printk (KERN_NOTICE "VA is %p\n", __va (physaddr));
12. 113     printk (KERN_NOTICE "Page at %p\n", virt_to_page (__va (physaddr)));
13. 114     if (!pfn_valid(pageframe))
14. 115         return NOPAGE_SIGBUS;
15. 116     pageptr = pfn_to_page(pageframe);
16. 117     printk (KERN_NOTICE "page->index = %ld mapping %p\n", pageptr->index, pageptr->mapping);
17. 118     printk (KERN_NOTICE "Page frame %ld\n", pageframe);
18. 119     get_page(pageptr);
19. 120     if (type)
20. 121         *type = VM_FAULT_MINOR;
21. 122     return pageptr;
22. 123}
```

106行，得到起始物理地址保存在offset中。

107行，得到address参数对应的物理地址，保存在physaddr中。

108行，得到address的物理地址对应的页帧号，保存在pageframe中。

116行，使用pfn_to_page函数，由页帧号返回对应的page结构指针保存在pageptr中。

119行，调用get_page增加pageptr指向页面的引用计数。

至此，simple模块的代码我们就分析完了。

[更多](#) 0

[上一篇](#) LDD3源码分析之与硬件通信&中断处理

[下一篇](#) S3C2410驱动分析之ADC通用驱动

顶

0

踩

0

主题推荐 [源码](#) [内存](#) [structure](#) [exception](#) [工作](#)

猜你在找

- [linux-3.2.36内核启动2-setup_arch中的内存初始化](#)
- [linux驱动中地址空间转换](#)
- [C++第9周\(春\)项目3 - 分数类](#)
- [vmlinux.lds解读](#)
- [如何在windows下面编译u-boot（原发于：2012-07-24](#)
- [Linux中IS_ERR\(\)函数的理解](#)

linux内核DMA内存分配

linux input输入子系统分析《二》：s3c2440的ADC简单

交叉编译环境下静态库动态库的加载

u-boot编译笔记

免费学习IT4个月,月薪12000

中国[官方授权]IT培训与就业示范基地,学成后名企直接招聘,月薪12000起!



查看评论

4楼 [frogcoder](#) 2013-06-05 15:28发表



107行的physaddr表示的是调整以后的物理地址？

Re: [wzw88486969](#) 2014-06-03 08:45发表



回复fjlhlong: unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
unsigned long physaddr = address - vma->vm_start + offset;
vma->vm_pgoff //页帧号
offset//页的开始地址
address - vma->vm_start //用户虚拟地址-用户虚拟开始地址=
在这一页里面的偏移
offset+address - vma->vm_start//, 得到address参数对应的物理地址

不知道我说的对不对，
那个offset的命名，有点让人。。。。。

3楼 [frogcoder](#) 2013-06-05 15:16发表



LZ,我对int simple_vma_nopage这个函数的107行不是很理解，106行的计算不是已经是物理地址了嘛？可以解释一下么？

2楼 [angelbosj](#) 2012-07-06 17:42发表



对照了LDD3，今天看完了你所有的博客。你真的很强。佩服，佩服。

1楼 [wjzws](#) 2012-06-02 03:55发表



哥们儿，辛苦了。解释地很细，受益匪浅。

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 [Java](#) [VPN](#) [Android](#) [iOS](#) [ERP](#) [IE10](#) [Eclipse](#) [CRM](#) [JavaScript](#) [Ubuntu](#) [NFC](#)
[WAP](#) [jQuery](#) [数据库](#) [BI](#) [HTML5](#) [Spring](#) [Apache](#) [Hadoop](#) [.NET](#) [API](#) [HTML](#) [SDK](#) [IIS](#)
[Fedora](#) [XML](#) [LBS](#) [Unity](#) [Splashtop](#) [UML](#) [components](#) [Windows Mobile](#) [Rails](#) [QEMU](#) [KDE](#)
[Cassandra](#) [CloudStack](#) [FTC](#) [coremail](#) [OPhone](#) [CouchBase](#) [云计算](#) [iOS6](#) [Rackspace](#)
[Web App](#) [SpringSide](#) [Maemo](#) [Compuware](#) [大数据](#) [aptech](#) [Perl](#) [Tornado](#) [Ruby](#) [Hibernate](#)
[ThinkPHP](#) [Spark](#) [HBase](#) [Pure](#) [Solr](#) [Angular](#) [Cloud Foundry](#) [Redis](#) [Scala](#) [Django](#)
[Bootstrap](#)

公司简介 | 招贤纳士 | 广告服务 | 银行汇款帐号 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320

京 ICP 证 070598 号

北京创新乐知信息技术有限公司 版权所有

江苏乐知网络技术有限公司 提供商务支持

Copyright © 1999-2014, CSDN.NET, All Rights Reserved

