

导航

[博客园](#)

[首页](#)

[新随笔](#)

[联系](#)

[订阅](#) [XML](#)

[管理](#)

<	2018年10月						>
	日	一	二	三	四	五	六
30	1	2	3	4	5	6	
7	8	9	10	11	12	13	
14	15	16	17	18	19	20	
21	22	23	24	25	26	27	
28	29	30	31	1	2	3	
4	5	6	7	8	9	10	

公告

昵称: LoveFM

园龄: 7年

粉丝: 93

关注: 3

[+加关注](#)

linux设备驱动程序中的阻塞机制

阻塞与非阻塞是设备访问的两种方式。在写阻塞与非阻塞的驱动程序时，经常用到等待队列。

一、阻塞与非阻塞

阻塞调用是指调用结果返回之前，当前线程会被挂起，函数只有在得到结果之后才会返回。

非阻塞指不能立刻得到结果之前，该函数不会阻塞当前进程，而会立刻返回。

对象是否处于阻塞模式和函数是不是阻塞调用有很强的相关性，但并不是一一对应的。阻塞对象上可以有非阻塞的调用方式，我们可以通过一定的API去轮询状态，在适当的时候调用阻塞函数，就可以避免阻塞。而对于非阻塞对象，调用的函数也可以进入阻塞调用。函数select()就是这样一个例子。

二、等待队列

在linux设备驱动程序中，阻塞进程可以使用等待队列来实现。

在内核中，等待队列是有很多用处的，尤其是在**中断处理**，**进程同步**，**定时**等场合，可以使用**等待队列实现阻塞进程的唤醒**。它以队列为基础数据结构，与进程调度机制紧密结合，能够用于实现内核中的异步事件通知机制，同步对系统资源的访问。

1、等待队列的实现：

在linux中，等待队列的结构如下：

```
struct __wait_queue_head {
    spinlock_t lock; //自旋锁，用来对task_list链表起保护作用，实现了对等待队列的互斥访问
    struct list_head task_list; //用来存放等待的进程
};
typedef struct __wait_queue_head wait_queue_head_t;
```

统计

随笔 - 23

文章 - 0

评论 - 21

引用 - 0

搜索

[找找看](#)

[谷歌搜索](#)

常用链接

[我的随笔](#)

[我的评论](#)

[我的参与](#)

[最新评论](#)

[我的标签](#)

我的标签

[ADS1.2\(1\)](#)

[NFS\(1\)](#)

[probe\(1\)](#)

[volatile\(1\)](#)

随笔分类

[ARM\(3\)](#)

[C/C++语言\(2\)](#)

[Linux内核分析OS\(2\)](#)

[Linux驱动开发\(7\)](#)

[Linux系统管理\(2\)](#)

2、等待队列的使用

(1) 定义和初始化等待队列:

```
wait_queue_head_t wait; //定义等待队列  
init_waitqueue_head(&wait); //初始化等待队列
```


定义并初始化等待队列:

```
#define DECLARE_WAIT_QUEUE_HEAD(name) wait_queue_head_t name =  
__WAIT_QUEUE_HEAD_INITIALIZER(name)
```

(2) 添加或移除等待队列:

```
void add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait); //将等待队列元  
素wait添加到等待队列头q所指向的等待队列链表中。  
void remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
```

(3) 等待事件:



```
wait_event(wq, condition); //在等待队列中睡眠直到condition为真。  
wait_event_timeout(wq, condition, timeout);  
wait_event_interruptible(wq, condition);  
wait_event_interruptible_timeout(wq, condition, timeout);  
/*  
*   queue:作为等待队列头的等待队列被唤醒  
*   conditon: 必须满足, 否则阻塞
```

Linux应用程序
嵌入式软件(1)
数据结构与算法(6)

随笔档案

2011年12月 (11)
2011年11月 (12)

最新评论

1. Re:Linux设备驱动之Ioctl控制

在编译用户态的测试程序的时候,
可以找到memdev.h中include的
linux/ioctl.h么?

--sherlock_wang

2. Re:linux设备驱动程序中的阻塞机制

问楼主一个问题, 内核会为驱动程序
创建一个进程运行环境吗, 换句
话, 驱动程序参与调度吗?

--UpInTheMornig

3. Re:如何计算Nand Flash要传入的行地址和列地址

看到你这里的文章, 估计另外一篇
计算地址的方法是从你这拷贝过去
的, 拷过去又瞎写, 什么第25页,
靠, 算了半天都对不上, 结果是第
64页。然后又看到一篇文章 里面
讲的地址换算好像也有点合理, 但
换算的结果又对不.....

--文少清

4. Re:linux设备驱动程序之简单字符设备驱动

老师您好! 我是在线IT教育-麦子
学院 () 的讲师顾问。从您的博

```
*      timeout和conditon相比, 有更高优先级
*/
```



(4) 睡眠:



```
sleep_on(wait_queue_head_t *q);
interruptible_sleep_on(wait_queue_head_t *q);
/*
sleep_on作用是把目前进程的状态置成TASK_UNINTERRUPTIBLE,直到资源可用, q引导的等待队列
被唤醒。
interruptible_sleep_on作用是一样的, 只不过它把进程状态置为TASK_INTERRUPTIBLE
*/
```



(5) 唤醒等待队列:

```
//可唤醒处于TASK_INTERRUPTIBLE和TASK_UNINTERRUPTIBLE状态的进程:
#define wake_up(x) __wake_up(x, TASK_NORMAL, 1, NULL)

//只能唤醒处于TASK_INTERRUPTIBLE状态的进程
#define wake_up_interruptible(x) __wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)
```

客了解到您, 钦佩您在Linux领域的优秀希望和您合作一些Linux方面的课程, 可以吗? 老师利用每周几个小时的时间就能向中国年轻人分.....

--麦99

5. Re:求子数组的最大和

___<3>

@无所谓了吧引用@zengnjin是不对, 他是从第一个数开始找的子集的最大数, 但是可能从中间开始的, 代码有问题我一开始也感觉不对劲, 后来一想, 没有漏洞的。。

(反证法) 最大子集如果是从中间开始的, 则说明.....

--校园

阅读排行榜

1. Linux设备驱动之Ioctl控制(52938)
2. linux设备驱动程序之简单字符设备驱动(38843)
3. ubuntu 10.04下的配置tftp服务器(13198)
4. Linux高级字符设备之Poll操作(10129)
5. Linux设备驱动之mmap设备操作(8533)

评论排行榜

1. Linux设备驱动之Ioctl控制(7)
2. linux设备驱动程序之简单字符设备驱动(7)

三、操作系统中睡眠、阻塞、挂起的区别形象解释

首先这些术语都是对于线程来说的。对线程的控制就好比控制了一个雇工为你干活。你对雇工的控制是通过编程来实现的。

挂起线程的意思就是你对主动对雇工说：“你睡觉去吧，用着你的时候我主动去叫你，然后接着干活”。

使**线程睡眠**的意思就是你主动对雇工说：“你睡觉去吧，某时某刻过来报到，然后接着干活”。

线程阻塞的意思就是，你突然发现，你的雇工不知道在什么时候没经过你允许，自己睡觉呢，但是你不能怪雇工，肯定你这个雇主没注意，本来你让雇工扫地，结果扫帚被偷了或被邻居家借去了，你又没让雇工继续干别的活，他就只好睡觉了。至于扫帚回来后，雇工会不会知道，会不会继续干活，你不用担心，雇工一旦发现扫帚回来了，他就会自己去干活的。因为雇工受过良好的培训。这个培训机构就是操作系统。

四、阻塞与非阻塞操作

阻塞操作是指在执行设备操作时若不能获得资源则挂起进程，直到满足可操作的条件后在进行操作。

非阻塞操作的进程在不能进行设备操作时并不挂起，它或者被放弃，或者不停的查询，直到可以进行操作为止。

回顾简单字符设备驱动，我们看到如何实现 read 和 write 方法。在此，但是，我们跳过了一个重要的问题：**一个驱动当它无法立刻满足请求应当如何响应？**一个对 read 的调用可能当没有数据时到来，而以后会期待更多的数据。或者一个进程可能试图写，但是你的设备没有准备好接受数据，因为你的输出缓冲满了。调用进程往往不关心这种问题；程序员只希望调用 read 或 write 并且使调用返回，在必要的工作已完成。这样，在这样的情形中，你的驱动应当(缺省地)阻塞进程，使它进入睡眠直到请求可继续。

3. 求子数组的最大和___<3>
(3)

4. 如何计算Nand Flash要传入的行地址和列地址(2)

5. s3c2440存储控制器和地址以及启动的理解(1)

推荐排行榜

1. linux设备驱动程序之简单字符设备驱动(13)

2. Linux设备驱动之Ioctl控制(6)

3. S3C2440的LCD编程(4)

4. Linux设备驱动之I/O端口与I/O内存(3)

5. Linux设备驱动之mmap设备操作(3)

在我们看全功能的 read 和 write 方法的实现之前, 我们触及的最后一点是决定何时使进程睡眠.

(1) 阻塞型驱动中, read实现方式: 如果一个进程调用 read 但是没有数据可用, 这个进程必须阻塞. 这个进程在有数据达到时被立刻唤醒, 并且那个数据被返回给调用者, 即便小于在给方法的 count 参数中请求的数量.

(2) 阻塞型驱动中, write实现方式: 如果一个进程调用 write 并且在缓冲中没有空间, 这个进程必须阻塞, 并且它必须在一个与用作 read 的不同的等待队列中. 当一些数据被写入硬件设备, 并且在输出缓冲中的空间变空闲, 这个进程被唤醒并且写调用成功, 尽管数据可能只被部分写入如果在缓冲只没有空间给被请求的 count 字节.

(3) 有时要求一个操作不阻塞, 即便它不能完全地进行下去. 应用程序元可以调用 filp->f_flags 中的 O_NONBLOCK 标志来人为的设置读写操作为非阻塞方式. 这个标志定义于 <linux/fcntl.h>, 被 <linux/fs.h> 自动包含.

五、阻塞型驱动测试程序:

1.memdev.h

```
#ifndef _MEMDEV_H_
#define _MEMDEV_H_

#ifndef MEMDEV_MAJOR
#define MEMDEV_MAJOR 0 /*预设的mem的主设备号*/
#endif
```

```
#ifndef MEMDEV_NR_DEVS
#define MEMDEV_NR_DEVS    /*设备数*/
#endif
```

```
#ifndef MEMDEV_SIZE
#define MEMDEV_SIZE  0
#endif
```



/*mem设备描述结构体*/

```
struct mem_dev
```

```
{
```

```
    char *data;
```

```
    unsigned long size;
```

```
    wait_queue_head_t inq;
```

```
};
```

```
#endif /* _MEMDEV_H_ */
```



2.memdev.c



```
#include linux/module h
#include linux/types h
#include linux/fs h
#include linux/errno h
#include linux/mm h
#include linux/sched h
#include linux/init h
#include linux/cdev h
#include asm/io h
#include asm/system h
#include asm/uaccess h
```

```
#include "memdev h"
```



```
static mem_ma or = MEMDEV_MAJOR;
bool have_data = false; /*表明设备有足够数据可供读*/
```

```
module_param(mem_ma or, int, S_IRU O);
```

```
struct mem_dev *mem_devp; /*设备结构体指针*/
```

```
struct cdev cdev;
```

```
/*文件打开函数*/
```

```
int mem_open(struct inode *inode, struct file *filp)
```

```
{
```

```
    struct mem_dev *dev;
```

```
    /*获取次设备号*/
```

```

    int num = MINOR(inode->i_rdev);

    if (num == MEMDEV_NR_DEVS)
        return ENODEV;

    dev = &mem_devp[num];

    /*将设备描述结构指针赋值给文件私有数据指针*/
    filp->private_data = dev;

    return 0;
}

/*文件释放函数*/
int mem_release(struct inode *inode, struct file *filp)
{
    return 0;
}

/*读函数*/
static ssize_t mem_read(struct file *filp, char __user *buf,
    ssize_t count, loff_t *ppos)
{
    unsigned long p = *ppos;
    unsigned int count = count;
    int ret = 0;
    struct mem_dev *dev = filp->private_data; /*获得设备结构体指针*/

    /*判断读位置是否有效*/
    if (p >= MEMDEV_SIZE)
        return 0;

```



```
if (count < MEMDEV_SIZE)
    count = MEMDEV_SIZE - p;

while (have_data) /* 没有数据可读，考虑为什么不用if，而用while，中断信号唤醒 */
{
    if (fcntl(fd, F_GETFL) & O_NONBLOCK)
        return EAGAIN;

    wait_event_interruptible(dev_inq, have_data);
}

/*读数据到用户空间*/
if (copy_to_user(buf, (void*)(dev_data + p), count))
{
    ret = EFAULT;
}
else
{
    *ppos = count;
    ret = count;

    printk(KERN_INFO "read %d bytes(s) from %d\n", count, p);
}

have_data = false; /* 表明不再有数据可读 */
return ret;
}

/*写函数*/
```

```
static ssize_t mem_write(struct file *filp, const char __user *buf, ssize_t
size, loff_t *ppos)
{
    unsigned long p = *ppos;
    unsigned int count = size;
    int ret = 0;
    struct mem_dev *dev = filp->private_data; /*获得设备结构体指针*/

    /*分析和获取有效的写长度*/
    if (p % MEMDEV_SIZE)
        return 0;
    if (count > MEMDEV_SIZE - p)
        count = MEMDEV_SIZE - p;

    /*从用户空间写入数据*/
    if (copy_from_user(dev->data + p, buf, count))
        ret = -E_AULT;
    else
    {
        *ppos += count;
        ret = count;

        printk(KERN_INFO "wrote %d bytes(s) from %d\n", count, p);
    }

    have_data = true; /* 有新的数据可读 */

    /* 唤醒读进程 */
    wake_up(&(dev->inq));
}
```

```

    return ret;
}

/* seek文件定位函数 */
static loff_t mem_llseek(struct file *filp, loff_t offset, int whence)
{
    loff_t newpos;

    switch(whence) {
        case 0: /* SEEK_SET */
            newpos = offset;
            break;

        case 1: /* SEEK_CUR */
            newpos = filp->f_pos + offset;
            break;

        case 2: /* SEEK_END */
            newpos = MEMDEV_SIZE + offset;
            break;

        default: /* can't happen */
            return EINVAL;
    }

    if ((newpos < 0) || (newpos > MEMDEV_SIZE))
        return EINVAL;

    filp->f_pos = newpos;
    return newpos;
}

```

```

}

/*文件操作结构体*/
static const struct file_operations mem_fops =
{
    owner = THIS_MODULE,
    llseek = mem_llseek,
    read = mem_read,
    write = mem_write,
    open = mem_open,
    release = mem_release,
};

/*设备驱动模块加载函数*/
static int memdev_init(void)
{
    int result;
    int i;

    dev_t devno = MKDEV(mem_major, 0);

    /* 静态申请设备号*/
    if (mem_major)
        result = register_chrdev_region(devno, 1, "memdev");
    else /* 动态分配设备号 */
    {
        result = alloc_chrdev_region(&devno, 0, 1, "memdev");
        mem_major = MAJOR(devno);
    }
}

```

```
if (result == 0)
    return result;

/*初始化cdev结构*/
cdev_init(&cdev, &mem_fops);
cdev.owner = THIS_MODULE;
cdev.ops = &mem_fops;

/* 注册字符设备 */
cdev_add(&cdev, MKDEV(mem_major, 0), MEMDEV_NR_DEVS);

/* 为设备描述结构分配内存*/
mem_devp = kmalloc(MEMDEV_NR_DEVS * sizeof(struct mem_dev), GFP_KERNEL);
if (!mem_devp) /*申请失败*/
{
    result = ENOMEM;
    goto fail_malloc;
}
memset(mem_devp, 0, sizeof(struct mem_dev));

/*为设备分配内存*/
for (i=0; i < MEMDEV_NR_DEVS; i++)
{
    mem_devp[i].size = MEMDEV_SIZE;
    mem_devp[i].data = kmalloc(MEMDEV_SIZE, GFP_KERNEL);
    memset(mem_devp[i].data, 0, MEMDEV_SIZE);

    /*初始化等待队列*/
    init_waitqueue_head(&(mem_devp[i].inq));
}
```

```

    return 0;

fail_malloc:
    unregister_chrdev_region(devno, 1);

    return result;
}

/*模块卸载函数*/
static void memdev_exit(void)
{
    cdev_del(&cdev);    /*注销设备*/
    kfree(mem_devp);    /*释放设备结构体内存*/
    unregister_chrdev_region(MKDEV(mem_major, 0), 1); /*释放设备号*/
}

MODULE_AUTHOR("David Lee");
MODULE_LICENSE("GPL");

module_init(memdev_init);
module_exit(memdev_exit);

```



3.app-write.c



```
#include <stdio.h>

int main()
{
    FILE *fp = NULL;
    char Buf[1024];

    /*打开设备文件*/
    fp = fopen("/dev/memdev0", "r+");
    if (fp == NULL)
    {
        printf("Open Dev memdev0 Error\n");
        return 1;
    }

    /*写入设备*/
    strcpy(Buf, "memdev is char dev ");
    printf("Write BU : %s\n", Buf);
    fwrite(Buf, sizeof(char), 1, fp);

    sleep(5);
    fclose(fp);

    return 0;
}
```



4.app-read.c



```
#include <stdio.h>

int main()
{
    FILE *fp = NULL;
    char Buf[1024];

    /*初始化Buf*/
    strcpy(Buf, "memdev is char dev ");
    printf("BU : s\n", Buf);

    /*打开设备文件*/
    fp = fopen("/dev/memdev0", "r");
    if (fp == NULL)
    {
        printf("Open memdev0 Error\n");
        return 1;
    }

    /*清除Buf*/
    strcpy(Buf, "Buf is NULL ");
    printf("Read BU 1: s\n", Buf);

    /*读出数据*/
    fread(Buf, sizeof(char), 1, fp);
```




```
/*检测结果*/
printf("Read BU : s n",Buf);

fclose(fp);

return 0;

}
```



分类: [Linux驱动开发](#)



LoveFM
关注 - 3
粉丝 - 93

[+加关注](#)

1

0

上一篇: [linux驱动程序中的并发控制](#)

下一篇: [Linux设备驱动之Ioctl控制](#)

posted on 2011-12-04 11:03 [LoveFM](#) 阅读(7013) 评论(1) [编辑](#) [收藏](#)

评论

#1楼 2016-07-06 19:25 UpInTheMornig

问楼主一个问题，内核会为驱动程序创建一个进程运行环境吗，换句话，驱动程序参与调度吗？

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

【推荐】超50万 C++源码：大型组态工控、电力仿真CAD与 IS源码库！

【推荐】华为云11.11普惠季 血拼风暴 一促即发

【拼团】腾讯云服务器拼团活动又双叒叕来了！

【推荐】腾讯云新注册用户域名抢购1元起

腾讯云
腾讯云AMD云服务器
节省IT成本30%
1核1G AMD机型0.57元/天起
立即抢购
The image is a promotional banner for Tencent Cloud's AMD Cloud Servers. It features a dark blue background with a subtle pattern of server racks and circuitry. At the top left is the Tencent Cloud logo. The main text is in white and yellow, highlighting the 'AMD Cloud Servers' and the '0.57元/天起' (starting from 0.57 yuan per day) price for a 1-core, 1GB model. A prominent blue button with white text says '立即抢购' (Buy Now). The bottom part of the banner shows stylized illustrations of server hardware.

最新IT新闻：

手握4000亿成就亚洲最大风投 高瓴是如何炼成的？

量子计算需熬十年冷 “BAT ”提前大搞军备竞赛？

微软新专利 使用游戏手柄或手势键入文本的新径向界面

嘀嗒剑走偏锋 C O宋中杰承诺决不碰专车、快车业务

我们试了下今日头条上线的这款电商app
[更多新闻...](#)



最新知识库文章:

[阿里云的这群疯子](#)

[为什么说 java 程序员必须掌握 Spring Boot ?](#)

[在学习中, 有一个比掌握知识更重要的能力](#)

[如何招到一个靠谱的程序员](#)

[一个故事看懂“区块链”](#)

[更多知识库文章...](#)

Powered by :

[博客园](#)

Copyright © LoveFM