

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： 75910

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）
- 2013年（3）
- 2012年（61）
- 2011年（66）
- 2010年（27）

Linux设备驱动程序学习（5）-高级字符驱动程序操作 [（2）阻塞型I/O和休眠]

2007-11-02 14:08:35

分类： LINUX

Linux设备驱动程序学习（5）

-高级字符驱动程序操作 [（2）阻塞型I/O和休眠]

这一部分主要讨论：[如果驱动程序无法立即满足请求，该如何响应？（65865346）](#)

一、休眠

进程被置为休眠，意味着它被标识为处于一个特殊的状态并且从调度器的运行队列中移走。这个进程将不被在任何CPU 上调度，即将不会运行。 直到发生某些事情改变了那个状态。安全地进入休眠的两条规则：

（1）永远不要在原子上下文中进入休眠，即[当驱动在持有一个自旋锁、seqlock或者 RCU 锁时不能睡眠；关闭中断也不能睡眠](#)。持有一个信号量时休眠是合法的，但你应当仔细查看代码：如果代码在持有一个信号量时睡眠，任何其他的等待这个信号量的线程也会休眠。因此发生在持有信号量时的休眠必须短暂，而且决不能阻塞那个将最终唤醒你的进程。

（2）当进程被唤醒，它并不知道休眠了多长时间以及休眠时发生什么；也不知道是否另有进程也在休眠等待同一事件，且那个进程可能在它之前醒来并获取了所等待的资源。所以不能对唤醒后的系统状态做任何的假设，并[必须重新检查等待条件来确保正确的响应](#)。

除非确信其他进程会在其他地方唤醒休眠的进程，否则也不能睡眠。使进程可被找到意味着：需要维护一个称为等待队列的数据结构。它是一个进程链表，其中包含了等待某个特定事件的所有进程。在 Linux 中， 一个等待队列由一个wait_queue_head_t 结构体来管理，其定义在<linux/wait.h>中。wait_queue_head_t 类型的数据结构非常简单：

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

它包含一个自旋锁和一个链表。这个链表是一个等待队列入口，它被声明做 wait_queue_t。wait_queue_head_t包含关于睡眠进程的信息和它想怎样被唤醒。

简单休眠（其实是高级休眠的宏）

Linux 内核中最简单的休眠方式是称为 wait_event的宏(及其变种)，它实现了休眠和进程等待的条件的检查。形式如下：

```
wait_event(queue, condition)/*不可中断休眠, 不推荐*/
wait_event_interruptible(queue, condition)/*推荐, 返回非零值意味着休眠被中断, 且驱动应返回 - ERESTARTSYS*/
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)
/*有限的时间的休眠; 若超时, 则不管条件为何值返回0, */
```

唤醒休眠进程的函数称为 wake_up，形式如下：

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

惯例：用 wake_up 唤醒 wait_event ；用 wake_up_interruptible 唤醒wait_event_interruptible。

简单休眠实验

模块程序链接：[sleepy](#)


模块测试程序链接：[sleepy-test](#)

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

实验现象：

```
[Tekkaman2440@SBC2440V4]#cd /lib/modules/
[Tekkaman2440@SBC2440V4]#insmod sleepy.ko
[Tekkaman2440@SBC2440V4]#cd /dev/
[Tekkaman2440@SBC2440V4]#cat /proc/devices
Character devices:
 1 mem
 2 pty
 3 ttyp
 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
180 usb
189 usb_device
204 s3c2410_serial
252 sleepy
253 usb_endpoint
254 rtc

Block devices:
 1 ramdisk
256 rfd
 7 loop
31 mtblock
93 nftl
96 inftl
179 mmc
[Tekkaman2440@SBC2440V4]#mknod -m 666 sleepy c 252 0
[Tekkaman2440@SBC2440V4]#cd /tmp/
[Tekkaman2440@SBC2440V4]#./sleepy_testr&
[Tekkaman2440@SBC2440V4]#./sleepy_testr&
[Tekkaman2440@SBC2440V4]#ps
PID Uid VSZ Stat Command
 1 root 1744 S init
 2 root SW< [kthreadd]
 3 root SWN [ksoftirqd/0]
 4 root SW< [watchdog/0]
 5 root SW< [events/0]
 6 root SW< [khelper]
59 root SW< [kblockd/0]
60 root SW< [ksuspend_usbd]
63 root SW< [khubd]
65 root SW< [kseriod]
77 root SW [pdflush]
78 root SW [pdflush]
79 root SW< [kswapd0]
80 root SW< [aio/0]
707 root SW< [mtblockd]
708 root SW< [nftld]
709 root SW< [inftld]
710 root SW< [rfdd]
```

```
742 root SW< [kpsmoused]
751 root SW< [kmmcd]
769 root SW< [rpciod/0]
778 root 1752 S -sh
779 root 1744 S init
781 root 1744 S init
783 root 1744 S init
787 root 1744 S init
799 root 1336 S ./sleepy_testr
800 root 1336 S ./sleepy_testr
802 root 1744 R ps
[Tekkaman2440@SBC2440V4]#./sleepy_testw
read code=0
write code=0
[2] + Done ./sleepy_testr
[Tekkaman2440@SBC2440V4]#ps
PID Uid VSZ Stat Command
  1 root 1744 S init
  2 root SW< [kthreadd]
  3 root SWN [ksoftirqd/0]
  4 root SW< [watchdog/0]
  5 root SW< [events/0]
  6 root SW< [khelper]
 59 root SW< [kblockd/0]
 60 root SW< [ksuspend_usbd]
 63 root SW< [khubd]
 65 root SW< [kseriod]
 77 root SW [pdflush]
 78 root SW [pdflush]
 79 root SW< [kswapd0]
 80 root SW< [aio/0]
707 root SW< [mtdblockd]
708 root SW< [nftld]
709 root SW< [inftld]
710 root SW< [rfdd]
742 root SW< [kpsmoused]
751 root SW< [kmmcd]
769 root SW< [rpciod/0]
778 root 1752 S -sh
779 root 1744 S init
781 root 1744 S init
783 root 1744 S init
787 root 1744 S init
799 root 1336 S ./sleepy_testr
804 root 1744 R ps
[Tekkaman2440@SBC2440V4]#./sleepy_testw
write code=0
[Tekkaman2440@SBC2440V4]#read code=0

[1] + Done ./sleepy_testr
[Tekkaman2440@SBC2440V4]#ps
PID Uid VSZ Stat Command
  1 root 1744 S init
  2 root SW< [kthreadd]
  3 root SWN [ksoftirqd/0]
  4 root SW< [watchdog/0]
  5 root SW< [events/0]
  6 root SW< [khelper]
 59 root SW< [kblockd/0]
 60 root SW< [ksuspend_usbd]
 63 root SW< [khubd]
 65 root SW< [kseriod]
 77 root SW [pdflush]
 78 root SW [pdflush]
 79 root SW< [kswapd0]
```

```

80 root SW< [aio/0]
707 root SW< [mtdblockd]
708 root SW< [nftld]
709 root SW< [inftld]
710 root SW< [rfdd]
742 root SW< [kpsmoused]
751 root SW< [kmmcd]
769 root SW< [rpciod/0]
778 root 1752 S -sh
779 root 1744 S init
781 root 1744 S init
783 root 1744 S init
787 root 1744 S init
806 root 1744 R ps

```

阻塞和非阻塞操作

全功能的 `read` 和 `write` 方法涉及到进程可以决定是进行非阻塞 I/O 还是阻塞 I/O 操作。明确的非阻塞 I/O 由 `filp->f_flags` 中的 `O_NONBLOCK` 标志来指示（定义再 `<linux/fcntl.h>`，被 `<linux/fs.h>` 自动包含）。浏览源码，会发现 `O_NONBLOCK` 的另一个名字：`O_NDELAY`，这是为了兼容 System V 代码。`O_NONBLOCK` 标志缺省地被清除，因为等待数据的进程的正常行为只是睡眠。

其实不一定只有 `read` 和 `write` 方法有阻塞操作，`open` 也可以有阻塞操作。后面会见到。而我的项目有一个和 CPLD 的接口的驱动，我决定要在 `ioctl` 中使用阻塞。

以下是后面的 `scullpipe` 实验的有关阻塞的代码，我觉得写得很好，先结合书上的介绍看看吧：

```

while (dev->rp == dev->wp)      { /* nothing to read */
    up(&dev->sem); /* release the lock */
    if (filp->f_flags & O_NONBLOCK)
        return -EAGAIN;
    PDEBUG("\'%s\' reading: going to sleep\n", current->comm);
    if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
        return -ERESTARTSYS; /* signal: tell the fs layer to handle it */ /* otherwise loop,
but first reacquire the lock */
    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
} /* ok, data is there, return something */
.....

```

高级休眠

步骤：

- （1）分配和初始化一个 `wait_queue_t` 结构，随后将其添加到正确的等待队列。
- （2）设置进程状态，标记为休眠。在 `<linux/sched.h>` 中定义有几个任务状态：`TASK_RUNNING` 意思是进程能够运行。有 2 个状态指示一个进程是在睡眠：`TASK_INTERRUPTIBLE` 和 `TASK_UNINTERRUPTIBLE`。2.6 内核的驱动代码通常不需要直接操作进程状态。但如果需要这样做使用的代码是：

```
void set_current_state(int new_state);
```

在老的代码中，你常常见到如此的东西：`current->state = TASK_INTERRUPTIBLE`；但是象这样直接改变 `current` 是不推荐的，当数据结构改变时这样的代码将会失效。通过改变 `current` 状态，只改变了调度器对待进程的方式，但进程还未让出处理器。

- （3）最后一步是放弃处理器。但必须先检查进入休眠的条件。如果不做检查会引入竞态：如果在忙于上面的这个过程时有其他的线程刚刚试图唤醒你，你可能错过唤醒且长时间休眠。因此典型的代码如下：

```
if (!condition)
    schedule();
```

如果代码只是从 `schedule` 返回，则进程处于 `TASK_RUNNING` 状态。如果不需睡眠而跳过对 `schedule` 的调用，必须将任务状态重置为 `TASK_RUNNING`，还必要从等待队列中去除这个进程，否则它可能被多次唤醒。

手工休眠

```

/* （1）创建和初始化一个等待队列。常由宏定义完成:*/
DEFINE_WAIT(my_wait);
/*name 是等待队列入口项的名字，也可以用2步来做:*/
wait_queue_t my_wait;
init_wait(&my_wait);

```

```

/*常用的做法是放一个 DEFINE_WAIT 在循环的顶部，来实现休眠。*/

/* (2) 添加等待队列入口到队列，并设置进程状态:*/
void prepare_to_wait(wait_queue_head_t *queue, wait_queue_t *wait, int state);
/*queue 和 wait 分别是等待队列头和进程入口。state 是进程的新状态: TASK_INTERRUPTIBLE(可中断休眠, 推荐)或TASK_UNINTERRUPTIBLE(不可中断休眠, 不推荐)。*/

/* (3) 在检查确认仍然需要休眠之后调用 schedule*/
schedule();

/* (4) schedule 返回，就到了清理时间:*/
void finish_wait(wait_queue_head_t *queue, wait_queue_t *wait);

```

认真地看简单休眠中的 `wait_event(queue, condition)` 和 `wait_event_interruptible(queue, condition)` 底层源码会发现，其实它们只是手工休眠中的函数的组合。所以怕麻烦的话还是用`wait_event`比较好。

独占等待

当一个进程调用 `wake_up` 在等待队列上，所有的在这个队列上等待的进程被置为可运行的。这在许多情况下是正确的做法。但有时，可能只有一个被唤醒的进程将成功获得需要的资源，而其余的将再次休眠。这时如果等待队列中的进程数目大，这可能严重降低系统性能。为此，内核开发者增加了一个“独占等待”选项。它与一个正常的睡眠有 2 个重要的不同：

- (1) 当等待队列入口设置了 `WQ_FLAG_EXCLUSIVE` 标志，它被添加到等待队列的尾部；否则，添加到头部。
- (2) 当 `wake_up` 被在一个等待队列上调用，它在唤醒第一个有 `WQ_FLAG_EXCLUSIVE` 标志的进程后停止唤醒。但内核仍然每次唤醒所有的非独占等待。

采用独占等待要满足 2 个条件：

- (1) 希望对资源进行有效竞争；
- (2) 当资源可用时，唤醒一个进程就足够来完全消耗资源。

使一个进程进入独占等待，可调用：

```
void prepare_to_wait_exclusive(wait_queue_head_t *queue, wait_queue_t *wait, int state);
```

注意：无法使用 `wait_event` 和它的变体来进行独占等待。

唤醒的相关函数

很少会需要调用`wake_up_interruptible` 之外的唤醒函数，但为完整起见，这里是整个集合：

```

wake_up(wait_queue_head_t *queue);
wake_up_interruptible(wait_queue_head_t *queue);
/*wake_up 唤醒队列中的每个非独占等待进程和一个独占等待进程。wake_up_interruptible 同样，除了它跳过处于不可中断休眠的进程。它们在返回之前，使一个或多个进程被唤醒、被调度(如果它们被从一个原子上下文调用，这就不会发生)。*/

wake_up_nr(wait_queue_head_t *queue, int nr);
wake_up_interruptible_nr(wait_queue_head_t *queue, int nr);
/*这些函数类似 wake_up，除了它们能够唤醒多达 nr 个独占等待者，而不只是一个。注意传递 0 被解释为请求所有的互斥等待者都被唤醒*/

wake_up_all(wait_queue_head_t *queue);
wake_up_interruptible_all(wait_queue_head_t *queue);
/*这种 wake_up 唤醒所有的进程，不管它们是否进行独占等待(可中断的类型仍然跳过在做不可中断等待的进程)*/

wake_up_interruptible_sync(wait_queue_head_t *queue);
/*一个被唤醒的进程可能抢占当前进程，并且在 wake_up 返回之前被调度到处理器。但是，如果你需要不要被调度出处理器时，可以使用 wake_up_interruptible 的“同步”变体。这个函数最常用在调用者首先要完成剩下的少量工作，且不希望被调度出处理器时。*/

```

poll 和 select

当应用程序需要进行对多文件读写时，若某个文件没有准备好，则系统会处于读写阻塞的状态，并影响了其他文件的读写。为了避免这种情况，在必须使用多输入输出流又不想阻塞在它们任何一个上的应用程序常将非阻塞 I/O 和 `poll` (System V)、`select` (BSD Unix)、`epoll` (linux 2.5.45 开始) 系统调用配合使用。当 `poll` 函数返回时，会给出一个文件是否可读写的标志，应用程序根据不同的标志读写相应的文件，实现非阻塞的读写。这些系统调用功能相同：允许进程来决定它是否可读或写一个或多个文件而不阻塞。这些调用也可阻塞进程直到任何一个给定集合的文件描述符可用来读或写。这些调用都需要来自设备驱动中 `poll` 方法的支持，`poll` 返回不同的标志，告诉主进程文件是否可以读写，其原型（定义在 `<linux/poll.h>`）：

```
unsigned int (*poll) (struct file *filp, poll_table *wait);
```

实现这个设备方法分两步：

1. 在一个或多个可指示查询状态变化的等待队列上调用 poll_wait。如果没有文件描述符可用来执行 I/O，内核使这个进程在等待队列上等待所有的传递给系统调用的文件描述符。驱动通过调用函数 poll_wait增加一个等待队列到 poll_table 结构，原型：

```
void poll_wait (struct file *, wait_queue_head_t *, poll_table *);
```

2. 返回一个位掩码：描述可能不必阻塞就立刻进行的操作，几个标志(通过 <linux/poll.h> 定义)用来指示可能的操作：

标志	含义
POLLIN	如果设备无阻塞的读，就返回该值
POLLRDNORM	通常的数据已经准备好，可以读了，就返回该值。通常的做法是会返回（POLLIN POLLRDNORM）
POLLRDBAND	如果可以从设备读出带外数据，就返回该值，它只可在linux内核的某些网络代码中使用，通常不用在设备驱动程序中
POLLPRI	如果可以无阻塞的读取高优先级（带外）数据，就返回该值，返回该值会导致select报告文件发生异常，以为select把带外数据当作异常处理
POLLHUP	当读设备的进程到达文件尾时，驱动程序必须返回该值，依照select的功能描述，调用select的进程被告知进程时可读的。
POLLERR	如果设备发生错误，就返回该值。
POLLOUT	如果设备可以无阻塞地些，就返回该值
POLLWRNORM	设备已经准备好，可以写了，就返回该值。通常地做法是（POLLOUT POLLNORM）
POLLWRBAND	于POLLRDBAND类似

考虑 poll 方法的 scullpipe 实现：

```
static unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
    struct scull_pipe *dev = filp->private_data;
    unsigned int mask = 0;
    /*
     * The buffer is circular; it is considered full
     * if "wp" is right behind "rp" and empty if the
     * two are equal.
     */
    down(&dev->sem);
    poll_wait(filp, &dev->inq, wait);
    poll_wait(filp, &dev->outq, wait);
    if (dev->rp != dev->wp)
        mask |= POLLIN | POLLRDNORM; /* readable */
    if (spacefree(dev))
        mask |= POLLOUT | POLLWRNORM; /* writable */
    up(&dev->sem);
    return mask;
}
```

与read 和write 的交互

正确实现poll调用的规则：

从设备读取数据：

- （1）如果在输入缓冲中有数据，read 调用应当立刻返回，即便数据少于应用程序要求的，并确保其他的数据会很快到达。 如果方便，可一直返回小于请求的数据，但至少返回一个字节。在这个情况下，poll 应当返回 POLLIN|POLLRDNORM。
- （2）如果在输入缓冲中无数据，read默认必须阻塞直到有一个字节。若O_NONBLOCK 被置位，read 立刻返回 -EAGAIN 。在这个情况下，poll 必须报告这个设备是不可读（清零POLLIN|POLLRDNORM）的直到至少一个字节到达。

（3）若处于文件尾，不管是否阻塞，`read` 应当立刻返回0，且`poll` 应该返回POLLHUP。

向设备写数据

（1）若输出缓冲有空间，`write` 应立即返回。它可接受小于调用所请求的数据，但至少必须接受一个字节。在这种情况下，`poll`应返回 POLLOUT|POLLWRNORM。

（2）若输出缓冲是满的，`write`默认阻塞直到一些空间被释放。若 `O_NONBLOCK` 被设置，`write` 立刻返回一个 `-EAGAIN`。在这些情况下，`poll` 应当报告文件是不可写的（清零POLLOUT|POLLWRNORM）。若设备不能接受任何多余数据，不管是否设置了 `O_NONBLOCK`，`write` 应返回 `-ENOSPC`（“设备上没有空间”）。

（3）永远不要让`write`在返回前等待数据的传输结束，即使`O_NONBLOCK` 被清除。若程序想保证它加入到输出缓冲中的数据被真正传送，驱动必须提供一个 `fsync` 方法。

刷新待处理输出

若一些应用程序需要确保数据被发送到设备，就实现必须`fsync` 方法。对 `fsync` 的调用只在设备被完全刷新时（即输出缓冲为空）才返回，不管 `O_NONBLOCK` 是否被设置，即便这需要一些时间。其原型是：

```
int (*fsync) (struct file *file, struct dentry *dentry, int datasync);
```

底层数据结构

只要用户应用程序调用 `poll`、`select`、或`epoll_ctl`，内核就会调用这个系统调用所引用的所有文件的 `poll` 方法，并向他们传递同一个`poll_table`。 `poll_table` 结构只是构成实际数据结构的简单封装：

```
struct poll_table_struct;
/*
 * structures and helpers for f_op->poll implementations
 */
typedef void (*poll_queue_proc)(struct file *, wait_queue_head_t *, struct poll_table_struct *);

typedef struct poll_table_struct {
    poll_queue_proc qproc;
} poll_table;
```

对于 `poll`和 `select`系统调用，`poll_table` 是一个包含 `poll_table_entry` 结构内存页链表。

```
struct poll_table_entry {
    struct file * filp;
    wait_queue_t wait;
    wait_queue_head_t * wait_address;
};
```

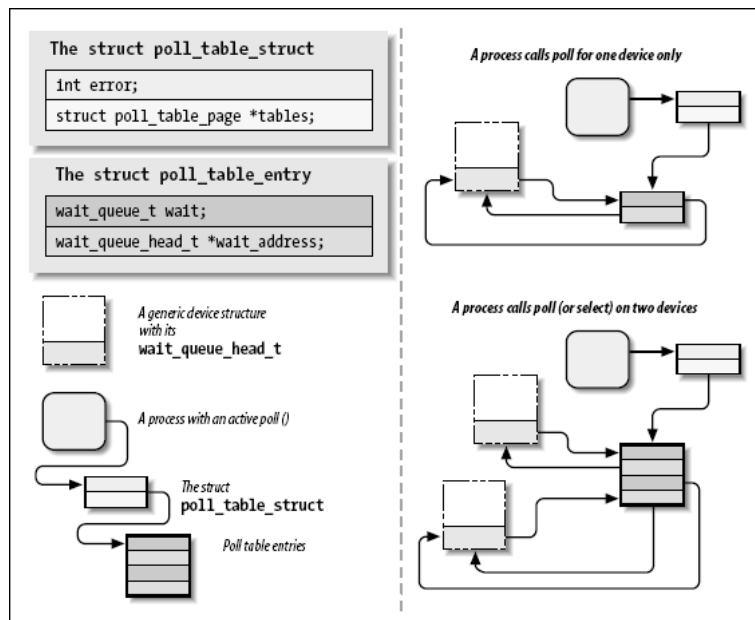
对 `poll_wait` 的调用有时还会将进程添加到给定的等待队列。整个的结构必须由内核维护，在 `poll` 或者 `select` 返回前，进程可从所有的队列中去除，。

如果被轮询的驱动没有一个驱动程序指明可进行非阻塞I/O，`poll` 调用会简单地睡眠，直到一个它所在的等待队列（可能许多）唤醒它。

当 `poll` 调用完成，`poll_table` 结构被重新分配，所有的之前加入到 `poll` 表的等待队列入口都会从表和它们的等待队列中移出。

```
struct poll_wqueues {
    poll_table pt;
    struct poll_table_page * table;
    int error;
    int inline_index;
    struct poll_table_entry inline_entries[N_INLINE_POLL_ENTRIES];
};

struct poll_table_page {
    struct poll_table_page * next;
    struct poll_table_entry * entry;
    struct poll_table_entry entries[0];
};
```

异步通知

通过使用异步通知，应用程序可以在数据可用时收到一个信号，而无需不停地轮询。

启用步骤:

(1) 它们指定一个进程作为文件的拥有者：使用 `fcntl` 系统调用发出 `F_SETOWN` 命令，这个拥有者进程的 ID 被保存在 `filp->f_owner`。目的：让内核知道信号到达时该通知哪个进程。

(2) 使用 `fcntl` 系统调用，通过 `F_SETFL` 命令设置 `FASYNC` 标志。

内核操作过程

1. `F_SETOWN`被调用时 `filp->f_owner`被赋值。
2. 当 `F_SETFL` 被执行来打开 `FASYNC`，驱动的 `fasync` 方法被调用. 这个标志在文件被打开时缺省地被清除。
3. 当数据到达时，所有的注册异步通知的进程都会被发送一个 `SIGIO` 信号。

Linux 提供的通用方法是基于一个数据结构和两个函数，定义在 `<linux/fs.h>`。

数据结构:

```
struct fasync_struct {
    int    magic;
    int    fa_fd;
    struct fasync_struct *fa_next; /* singly linked list */
    struct file *fa_file;
};
```

驱动调用的两个函数的原型:

```
int fasync_helper(int fd, struct file *filp, int mode, struct fasync_struct **fa);
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

当一个打开的文件的 `FASYNC` 标志被修改时，调用 `fasync_helper` 来从相关的进程列表中添加或删除文件。除了最后一个参数，其他所有参数都时被提供给 `fasync` 方法的相同参数并被直接传递。当数据到达时，`kill_fasync` 被用来通知相关的进程，它的参数是被传递的信号 (常常是 `SIGIO`) 和 `band` (几乎都是 `POLL_IN`)。

这是 `scullpipe` 实现 `fasync` 方法的:

```
static int scull_p_fasync(int fd, struct file *filp, int mode)
{
    struct scull_pipe *dev = filp->private_data;
    return fasync_helper(fd, filp, mode, &dev->async_queue);
}
```

当数据到达，下面的语句必须被执行来通知异步读者。因为对 `scullpipe` 读者的新数据通过一个发

出 write 的进程被产生, 这个语句出现在 scullpipe 的 write 方法中:

```
if (dev->async_queue)
    kill_fasync(&dev->async_queue, SIGIO, POLL_IN); /* 注意, 一些设备也针对设备可写而实现了异步
```

当文件被关闭时必须调用fasync 方法, 来从活动的异步读取进程列表中删除该文件。尽管这个调用仅当 filp->f_flags 被设置为 FASYNC 时才需要, 但不管什么情况, 调用这个函数不会有问题, 并且是普遍的实现方法。 以下是 scullpipe 的 release 方法的一部分:

```
/* remove this filp from the asynchronously notified filp's */
scull_p_fasync(-1, filp, 0);
```

异步通知使用的数据结构和 struct wait_queue 几乎相同, 因为他们都涉及等待事件。区别异步通知用 struct file 替代 struct task_struct. 队列中的 file 用获取 f_owner, 一边给进程发送信号。

scullpipe的实验 (poll和fasync方法的实现):

模块程序链接: **scullpipe**

模块测试程序链接: **scullpipe-test**

ARM9实验板的实验现象是:

```
[Tekkaman2440@SBC2440V4]#cd /lib/modules/
[Tekkaman2440@SBC2440V4]#insmod pipe.ko
[Tekkaman2440@SBC2440V4]#cat /proc/devices
Character devices:
 1 mem
 2 pty
 3 ttyp
 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
180 usb
189 usb_device
204 s3c2410_serial
252 pipe
253 usb_endpoint
254 rtc

Block devices:
 1 ramdisk
256 rfd
 7 loop
31 mtdblock
93 nftl
96 inftl
179 mmc
[Tekkaman2440@SBC2440V4]#cd /dev/
[Tekkaman2440@SBC2440V4]#
[Tekkaman2440@SBC2440V4]#cd /tmp/
[Tekkaman2440@SBC2440V4]#./pipe_test &
[Tekkaman2440@SBC2440V4]#open scullpipe0 !
open scullpipe1 !
SCULL_P_IOCTLSIZE : scull_p_buffer0=21 !
```

```
SCULL_P_IOCTLSIZE : scull_p_buffer1=21 !
close pipetest0 !
close pipetest1 !
reopen scullpipe0 !
reopen scullpipe1 !

[Tekkaman2440@SBC2440V4]#echo 12345678901234567890 > /dev/scullpipe0
[Tekkaman2440@SBC2440V4]#read from pipetest0 code=20
[0]=1 [1]=2 [2]=3 [3]=4 [4]=5
[5]=6 [6]=7 [7]=8 [8]=9 [9]=0
[10]=1 [11]=2 [12]=3 [13]=4 [14]=5
[15]=6 [16]=7 [17]=8 [18]=9 [19]=0
read from pipetest0 code=1
[0]=
[1]=2 [2]=3 [3]=4 [4]=5
[5]=6 [6]=7 [7]=8 [8]=9 [9]=0
[10]=1 [11]=2 [12]=3 [13]=4 [14]=5
[15]=6 [16]=7 [17]=8 [18]=9 [19]=0

[Tekkaman2440@SBC2440V4]#echo 12345678901234 > /dev/scullpipe1
[Tekkaman2440@SBC2440V4]#read from pipetest1 code=15
[0]=1 [1]=2 [2]=3 [3]=4 [4]=5
[5]=6 [6]=7 [7]=8 [8]=9 [9]=0
[10]=1 [11]=2 [12]=3 [13]=4 [14]=
[15]=6 [16]=7 [17]=8 [18]=9 [19]=0

[Tekkaman2440@SBC2440V4]#ps
PID Uid VSZ Stat Command
1 root 1744 S init
2 root SW< [kthreadd]
3 root SWN [ksoftirqd/0]
4 root SW< [watchdog/0]
5 root SW< [events/0]
6 root SW< [khelper]
59 root SW< [kblockd/0]
60 root SW< [ksuspend_usbd]
63 root SW< [khubd]
65 root SW< [kseriod]
77 root SW [pdflush]
78 root SW [pdflush]
79 root SW< [kswapd0]
80 root SW< [aio/0]
707 root SW< [mtdblockd]
708 root SW< [nftld]
709 root SW< [inftld]
710 root SW< [rfdd]
742 root SW< [kpsmoused]
751 root SW< [kmmcd]
769 root SW< [rpciod/0]
778 root 1752 S -sh
779 root 1744 S init
781 root 1744 S init
783 root 1744 S init
785 root 1744 S init
796 root 1344 S ./pipe_test
797 root 1744 R ps
[Tekkaman2440@SBC2440V4]#./asynctest &
[Tekkaman2440@SBC2440V4]#echo 12345678901234 > /dev/scullpipe0
[Tekkaman2440@SBC2440V4]#12345678901234
close pipetest0 !
exit !

[2] + Done ./asynctest
[Tekkaman2440@SBC2440V4]#cat /proc/scullpipe
```

```
Default buffersize is 21

Device 0: c3e18494
  Queues: c04flc34 c04flc34
  Buffer: c3dc6c08 to c3dc6c1d (21 bytes)
  rp c3dc6c17 wp c3dc6c17
  readers 1 writers 0

Device 1: c3e18528
  Queues: c04flc6c c04flc6c
  Buffer: c3dc6b38 to c3dc6b4d (21 bytes)
  rp c3dc6b47 wp c3dc6b47
  readers 1 writers 0

Device 2: c3e185bc
  Queues: c3e185bc c3e185bc
  Buffer: 00000000 to 00000000 (0 bytes)
  rp 00000000 wp 00000000
  readers 0 writers 0

Device 3: c3e18650
  Queues: c3e18650 c3e18650
  Buffer: 00000000 to 00000000 (0 bytes)
  rp 00000000 wp 00000000
  readers 0 writers 0
```

阅读(7874) | 评论(0) | 转发(34) |

上一篇: [转载]中国最致命的薄弱环节! (一个机械类毕业生的心声)

下一篇: Linux系统调用一fcntl函数详解

0

相关热门文章

- | | | |
|-----------------------|----------------------------|-----------------------|
| 云存储的优势有哪些 | linux 常见服务端口 | 移植 ushare 到开发板 |
| 私有云到底有什么用 | 【ROOTFS搭建】busybox的httpd... | 系统提供的库函数存在内存泄漏... |
| 企业私有云存储有哪些功能... | xmanager 2.0 for linux配置 | linux虚拟机 求教 |
| 常用MFC和API函数 | 什么是shell | 初学UNIX环境高级编程的, 关于... |
| mtd子系统-向block系统注册块... | linux socket的bug?? | chinaunix博客什么时候可以设... |

给主人留下些什么吧! ^^

评论热议

登录后评论。

[登录](#) [注册](#)

