一、信号量

举例子说明信号量作用：如果一个设备只能被一个进程打开，如何实现？

以前的驱动程序，像 led 设备，应用可以同时启动多个打开 led 设备，像下面一样操作：

./a.out&

./a.out&

./a.out &

&的意思是把这个进程加入到后台运行，你可以使用 jobs 查看当前后台进程运行情况，执行 jobs -l 查看即可。

如果想杀死这个进程有一下俩个选择：

jobs -l kill jib 号　　或　　ps kill -9 进程号

操作就产生了 3 个进程，这 3 个进程都可以随便对这个 led 设备进行操作。可能就造成控制灯冲突的情况。那如何解决这个问题？

先分析，为什么可以多进程对这个 led 进行操作。原因是 led 设备每次调用 open 函数都成功，成功就可以得到操作 led 文件描述符。

驱动代码的 open 函数：

```
int first_chardev_open (struct inode *inode, struct file *file)
{
    printk(KERN_EMERG"open() call\r\n");
    return 0 ;
}
```

这个驱动的驱动代码使应用程序每次调用 open 都成功。如果要实现只有一个进程可以打开，就要在 open 函数里做判断：如果 led 已经被打开，就返回错误码；如果没有被打开，就返回 0。

裸机编程方法：

```
int open_flag = 1; //添 加 一 个 全 局 变 量
int first_chardev_open (struct inode *inode, struct file *file)
{
    //第一次打开是成功，
    if(open_flag==0)
        return -EBUSY;
    open_flag --;
    printk(KERN_EMERG"open() call\r\n");
    return 0 ;
}
```

关闭文件时候执行 release 接口函数，所以也要相应的修改，关闭时要还原。

```
int first_chardev_close (struct inode *inode, struct file *file)
{
    open_flag = 1;
    printk(KERN_EMERG"release() call\r\n");
    return 0;
}
```

内核信号量结构：

```
struct semaphore
{
    raw_spinlock_t lock;
    unsigned int count;
        //信号值，只要这个值为正数，则信号可用，一般情况设置 0 或 1。
    struct list_head wait_list;
};
```

定义、初始化信号量相关 API:

```
#define DEFINE_SEMAPHORE(name) \
            struct semaphore name = __SEMAPHORE_INITIALIZER(name, 1)
```

这是一个静态定义方式。定义一个名字为 name,信号量值为 1 的信号量结构变量。

动态初始化函数：static inline void sema_init(struct semaphore *sem, int val)

参数 sem：要初始化信号量结构指针，

val：要设置到信号量结构 count 成员的值，也就是信号值。

例：

struct semaphore sem;

sema_init(&sem, 1);

以上两行等效：

DEFINE_SEMAPHORE(sem);

获取信号量相关 API:

A、extern void down(struct semaphore *sem);

阻塞地请求一个信号量,如果信号量大于 0，则可以马上返回，否则休眠，直到有其他进程释放信号量（把信号量的 count 修改为大于 0 的值）。

B、extern int __must_check down_interruptible(struct semaphore *sem);

阻塞地请求一个信号量,如果信号量大于 0，则可以马上返回，否则休眠，直到有其他进程释放信号量（把信号量的 count 修改为大于 0 的值）或者有中断信号产生。

C、extern int __must_check down_trylock(struct semaphore *sem);

非阻塞地请求一个信号量,如果信号量大于 0，则可以马上返回，如果小于 0，也返回，通过判断返回返回值判断是否成功得到了信号。

0：表成功得到了信号

1：没有得到信号量

释放信号量相关 API:

extern void up(struct semaphore *sem);

释放信号量，对信号量的 count 成员进行加 1 操作。

对前面程序进行修改，实现进程独占打开。

```
/* 定义一个初始值为 1 的信号量结构 */
DEFINE_SEMAPHORE(sem);

int first_chardev_open (struct inode *inode, struct file *file)
{
    down(&sem);
    return 0 ;
}
```

释放信号量：

```
int first_chardev_close (struct inode *inode, struct file *file)
{
    up(&sem);
    printk(KERN_EMERG"release() call\r\n");
    return 0;
}
```

二、自旋锁

自旋：自旋锁最多只能被一个可执行单元持有，自旋锁不会引起调用者睡眠，如果一个执行线程试图获得一个已经被持有的自旋锁，那么线程就会一直进行忙碌循环，一直等待下去，直到该自旋锁的保持者已经释放了锁。

定义关键性结构体：spinlock_t luck;

包含头文件#include <linux/spinlock.h>

1、初始化锁，使锁可以工作

spin_lock_init (&kuck);　　//通常在模块初始化的时候初始化锁；

2、加锁

A、spin_lock (&lock);

获取自旋锁 lock ，如果成功，立即获得锁，并马上返回，否则它将一直自旋在那里，直到该自旋锁的保持者释放。

B、spin_trylock (lock);

获取自旋锁 lock ，如果能立即获得锁，并返回真，否则立即返回假，它不会一直等待被释放。

3、解锁

spin_unlock (lock)　//释放自旋锁 lock，它与 spin_trylock 或 spin_lock 配对使用。

三、信号量和自旋锁的区别

信号量是睡眠锁，自旋锁是忙碌锁。

1、信号量可能允许多个持有者，而自旋锁在任何时候只能允许一个持有者；

当然也有信号量叫互斥信号量（只能有一个持有者），允许有多个持有者的信号量叫计数信号量。

2、信号量适合于保持时间长的情况，而自旋锁适合于保持时间非常短的情况；

在实际应用中，自旋锁控制的代码只有几行，而持有自旋锁的时间也一般不会超过两次上下文切换的时间，因为线程一旦要进行切换，就至少花费切出切入两次，自旋锁的占用时间如果远远长于两次上下文切换，我们就应该选择信号量。自旋锁通常用于在驱动程序中保护一段代码同时只有一个进程访问。

四、异步通知

信号机制 ---- 实现异步通知

它是操作系统实现的一种软中断机制（可以打断正在运行的进程）；

kill -9 pid (杀死一个 pid 进程)；

crtl + c 结束一个进程 = kill -2 pid（只能杀死前台进程）；

进程接收到信号 SIGINT，打断正在执行的代码，去调用操作系统预定义的信号 INT 的处理函数，这个处理函数就是释放进程资源退出进程。

ctrl + \ = kill -3 pid

在终端显示所有信号机制：

```
[root@localhost ~]# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
[root@localhost ~]#
```

实现异步通知：（从内核层直接发信号到应用层）

驱动程序运行在内核空间中，应用程序运行在用户空间中，两者是不能直接通信的。但在实际应用中，在设备已经准备好的时候，我们希望通知用户程序设备已经 ok，用户程序可以读取了，这样应用程序就不需要一直查询该设备的状态，从而节约了资源，这就是异步通知 fasync。

应用端：

1、因为是驱动端给应用端发信号，所以应用端首先要捕捉相应内核信号，并且在相应注册信号的函数内去读取键值。

例：signal(SIGIO,sig_fun);

```
void sig_fun(int signum)
{
    static unsigned char val = 0;
    read(fd,&val,1);
    printf("val =  %d\n",val);
}
```

2、fcntl(fd, F_SETOWN, getpid());//getpid()函数返回当前进程的进程号。指定一个进程作为文件的“属主(filp->owner)”，这样内核才知道信号要发给哪个进程；

3、设置文件标志，添加 **FASYNC** 标志 Oflags = fcntl(fd, F_GETFL);fcntl(fd, F_SETFL,Oflags | FASYNC);

//驱动想要发送信号的前提是设备文件的标志中有 **FASYNC**

配置以上三步，这样当内核执行到 kill_fasync 函数，用户空间 SIGIO 函数的处理函数就会被调用了。

驱动端：

1、定义 struct fasync_struct *mykey_fasync; 定义异步通知关键性结构体指针

2、实现设备操作中 file_operations 中的.fasync 函数。

int (*fasync) (int, struct file *, int);

因为内核异步通知的结构很复杂，系统给我们提供了一个函数 fasync_helper 来帮助我们初始化 struct fasync_struct 这个结构体，包括分配内存和设置属性。只要简单调用 fasync_helper 这个函数就可以了，fasync_helper 中的前三个参数都是由内核分配，直接传递过去即可，然后把自己定义的 struct fasync_struct 指针告诉给这个函数。

int fasync_helper(int fd, struct file * filp, int on, struct fasync_struct **fapp)

//注意最后一个参数是一个二级指针。

3、kill_fasync(&mykey_fasync,SIGIO,POLL_IN); //什么时候发送什么信号;

过程:

应用层: 捕捉信号 → 指定文件属组 → 设置文件标志

驱动层: 配置 file_operations 里面的 fasync 函数

→利用 fasync_helper 函数帮助填充→用 kill_fasync 函数发送信号

五、异步通知上层函数分析 --- fcntl 函数

参考: http://itlab.idcquan.com/linux/soft/881313.html

fcntl 函数作用: 根据文件描述词来操作文件的特性。

原型: #include <fcntl.h>

int fcntl (int fd, int cmd);

int fcntl (int fd, int cmd, long arg);

int fcntl (int fd, int cmd, struct flock *lock);

fcntl() 针对 (文件) 描述符提供控制。参数 fd 是被参数 cmd 操作 (如下面的描述) 的描述符。针对 cmd 的值, 判断 fcntl 是否能够接受第三个参数 int arg。

fcntl 函数有 5 种功能如下: 1.复制一个现有的描述符 (cmd=F_DUPFD)。

2.获得/设置文件描述符标记 (cmd=F_GETFD 或 F_SETFD)。

3.获得/设置文件状态标记 (cmd=F_GETFL 或 F_SETFL)。

4.获得/设置异步 I/O 所有权 (cmd=F_GETOWN 或 F_SETOWN)。

5.获得/设置记录锁 (cmd=F_GETLK,F_SETLK 或 F_SETLKW)。

以下是其详细的介绍:

A、cmd=F_DUPFD --- 复制一个现有的文件描述符, 第三个参数不用

例: fcntl(fd,F_DUPFD); --- 复制一个和 fd 性质相同的文件描述符并返回。

相同性质包含: 共享偏移量、相同读写方式、相同的打开状态等。

返回的 (文件) 描述符特点:

· 最小的大于或等于 arg 的一个可用的描述符;

· 与原始操作符一样的某对象的引用;

· 如果对象是文件(file)的话,返回一个新的描述符,这个描述符与 arg 共享相同的偏移量(offset);

· 相同的访问模式 (读, 写或读/写);

· 相同的文件状态标志 (如: 两个文件描述符共享相同的状态标志);

· 与新的文件描述符结合在一起的 close-on-exec 标志被设置成交叉式访问;

B、cmd=F_GETFD --- 获取文件的 close-on-exec 标志。

这个标志的意思是在当前进程调用 exec 函数族时, 当这个标志为 1 的时候代表新进程产生的时候已经关闭了此文件, 为 0 的时候没有关闭。不用第三个参数。

C、cmd=F_SETFD --- 设置文件的 close-on-exec 标志, 用第三个参数代表要设置的标志位;

D、cmd=F_GETFL --- 取得文件状态标志, 不用第三个参数;

E、cmd-F_SETFL --- 设置文件标志, 需要第三个参数;

原有的标志位如下:

1)、O_NONBLOCK:非阻塞 IO, 意思为 read 读取的时候无数据可读以及 write 写时不会阻塞, 会返回错误;

2)、O_APPEND: 强制 write 写的时候都写在最后, 追加的意思;

3)、O_DIRECT： 最小化或去掉 reading 和 writing 的缓存影响。系统将企图避免缓存你的读或写的数据。如果不能够避免缓存，那么它将最小化已经被缓存了的数据造成的影响。如果这个标志用的不够好，将大大的降低性能；

4)、O_ASYNC 当 I/O 可用的时候，允许 SIGIO 信号发送到进程组。

例：当有数据可以读的时候；

这些标志位设置之前一定要注意，先获取标志位，然后将你要设置的标志位或上原有的标志位，如果只是执行 F_SETFD 或 F_SETFL 命令，则原有的标志位会被你新设置的覆盖，这样文件只有你设置的标志位了。

F、cmd=F_GETOWN --- 取得当前正在接收 SIGIO 或者 SIGURG 信号的进程 id 或进程组 id，进程组 id 返回成负值（arg 被忽略）；

G、cmd=F_SETOWN --- 设置将要接收 SIGIO 或 SIGURG 信号的进程 id 或进程组 id，第三个参数为我们要设置的 ID。

H、cmd=F_GETLK --- 通过第三个参数 arg（一个指向 flock 的结构体）取得第一个阻塞 lock description 指向的的锁。取得的信息将覆盖传到 fcntl()的 flock 结构的信息。如果没有发现能够阻止本次锁（flock）生成的锁，这个结构将不被改变，除非锁的类型被设置成 F_UNLCK；

I、cmd=F_SETLK --- 按照指向结构体 flock 的指针的第三个参数 arg 所描述的锁的信息设置或者清除一个文件 segment 锁。

F_SETLK 被用来实现共享（读）锁（F_RDLCK）或独占（写）锁（F_WRLCK），同样可以去掉（F_UNLCK）这两种锁。如果共享锁或独占锁不能被设置，fcntl()将立即返回 EAGAIN；

J、cmd=F_SETLKW --- 除了共享锁或独占锁被其他的锁阻塞这种情况外，这个命令和 F_SETLK 是一样的。如果共享锁或独占锁被其他的锁阻塞，进程将等待，直到这个请求能够完成。

当 fcntl()正在等待文件的某个区域的时候捕捉到一个信号，如果这个信号没有被指定 SA_RESTART，fcntl()将被中断；

注：当一个共享锁被 set 到一个文件的某段的时候，其他进程可以 set 共享锁到这个段或这个段的一部分，但不能 set 独占锁到这段保护区域的任何部分。如果文件描述符没有以读的访问方式打开的话，共享锁的设置请求会失败；

独占锁阻止其他任何进程在这段保护区域任何位置设置共享锁或独占锁。如果文件描述符不是以写的访问方式打开的话，独占锁的请求会失败；

结构体 flock 的指针：

```
struct flcok
{
    short int l_type;   /* 锁定的状态*/
    //以下三个参数用于分段对文件加锁；
     //若对整个文件加锁，则：l_whence=SEEK_SET,  l_start=0,  l_len=0;
    short int l_whence;   /*决定 l_start 位置*/
    off_t l_start;          /*锁定区域的开头位置*/
    off_t l_len;            /*锁定区域的大小*/
    pid_t l_pid;            /*锁定动作的进程*/
};
```

参数 l_type： 有三种状态： F_RDLCK 建立一个供读取用的锁定；

F_WRLCK 建立一个供写入用的锁定；

F_UNLCK 删除之前建立的锁定；

参数 l_whence：有三种方式： SEEK_SET 以文件开头为锁定的起始位置；
SEEK_CUR 以目前文件读写位置为锁定的起始位置；
SEEK_END 以文件结尾为锁定的起始位置；

返回值：成功则返回 0，若有错误则返回-1，错误原因存于 errno；

六、fcntl 文件锁

有两种类型：建议性锁和强制性锁

建议性锁：每个使用上锁文件的进程都要检查是否有锁存在，当然还得尊重已有的锁。
内核和系统总体上都坚持不使用建议性锁，它们依靠程序员遵守这个规定。

强制性锁：由内核执行，当文件被上锁来进行写入操作时，在锁定该文件的进程释放该锁之前，
内核会阻止任何对该文件的读或写访问，每次读或写访问都得检查锁是否存在。

系统默认 fcntl 都是建议性锁，强制性锁是非 POSIX 标准的。如果要使用强制性锁，要使整个系统可以使用强制性锁，那么得需要重新挂载文件系统， mount 使用参数 -0 mand 打开强制性锁，或者关闭已加锁文件的组执行权限并且打开该文件的 set-GID 权限位。

建议性锁只在 cooperating processes 之间才有用，对 cooperating process 的理解是最重要的，它指的是会影响其它进程的进程或被别的进程所影响的进程。

举两个例子：

（1）我们可以同时在两个窗口中运行同一个命令，对同一个文件进行操作，
那么这两个进程就是 cooperating processes；

（2） cat file | sort， 那么 cat 和 sort 产生的进程就是使用了 pipe 的 cooperating processes；

使用 fcntl 文件锁进行 I/O 操作必须小心：进程在开始任何 I/O 操作前如何去处理锁，在对文件解锁前如何完成所有的操作，是必须考虑的。如果在设置锁 之前打开文件，或者读取该锁之后关闭文件，另一个进程就可能在上锁/解锁操作和打开/关闭操作之间的几分之一秒内访问该文件。当一个进程对文件加锁后，无 论它是否释放所加的锁，只要文件关闭，内核都会自动释放加在文件上的建议性锁（这也是建议性锁和强制性锁的最大区别）， 所以不要想设置建议性锁来达到永久不让别的进程访问文件的目的（强制性锁才可以）；强制性锁则对所有进程起作用。

fcntl 使用三个参数 F_SETLK/F_SETLKW， F_UNLCK 和 F_GETLK，来分别要求、释放、测试 record locks， record locks 是对文件一部分而不是整个文件的锁，这种细致的控制使得进程更好地协作以共享文件资源。fcntl 能够用于读取锁和写入锁，read lock 也叫 shared lock（共享锁）， 因为多个 cooperating process 能够在文件的同一部分建立读取锁；write lock 被称为 exclusive lock（排斥锁）， 因为任何时刻只能有一个 cooperating process 在文件的某部分上建立写入锁。如果 cooperating processes 对文件进行操作，那么它们可以同时对文件加 read lock， 在一个 cooperating process 加 write lock 之前，必须释放别的 cooperating process 加在该文件的 read lock 和 wrtie lock， 也就是说，对于文件只能有一个 write lock 存在，read lock 和 wrtie lock 不能共存。

fcntl 的返回值与命令有关。如果出错，所有命令都返回－1，如果成功则返回某个其他值。
下列三个命令有特定返回值：

F_DUPFD：返回新的文件描述符；

F_GETFD：返回相应标志；

F_GETFL 以及 F_GETOWN 返回一个正的进程 ID 或负的进程组 ID。

综述：

异步通知就是驱动层可以给上层发送一个信号，可以通过此操作告诉应用层某个操作完成了；

驱动层：

此时驱动层应该填充 file_operations 里面的 fasync 函数。填充这个函数起始很复杂，内核给我们封装了一个帮助函数 fasync_helper 函数，我们直接调 用即可。然后我们要发送信号调用 kill_fasync 函数即可。

应用层：

首先你要注册捕捉信号函数，然后你要把进程号告诉驱动层 fcntl(fd,F_SETOWN,getpid());

获取当前的标志位 Oflags = fcntl(fd,F_GETFL);

更改这个标志位 fcntl(fd,F_SETFL,Oflags|FASYNC);

完成以上驱动层和应用层的填充就可以实现异步通知了。