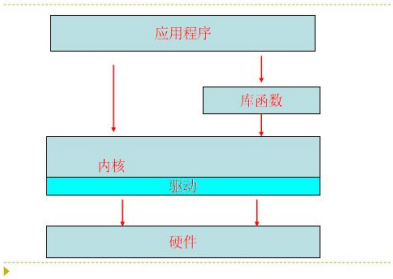
1. 设备驱动定义

设备驱动程序是一个**软件层**，该软件层使**硬件响应预定义好的编程接口**，我们已经熟悉了这种接口，它由一组控制**设备的 VFS 函数（open,read,lseek,ioctl 等）组成**，这些函数实际实现**由设备驱动程序全权负责**。



2. 模块

2.1 模块定义：可在运行时添加到内核中的代码被称为“模块”。

2.2 模块优点：

（1）模块化方法

因为任何模块都可以在运行时被链接和解除链接，因此，系统程序员必须提出**明确定义的软件接口**以**访问由模块处理的数据结构**，这使得**开发新模块变得容易**；

（2）平台无关性

即使模块**依赖于某些特殊的硬件特点**，但它**不依赖于某个固定的硬件平台**；

（3）节省内存使用

当**需要**模块功能时，把它**链接**到正在运行的内核中，否则，将该模块解除链接；

（4）无性能损失

模块的目标代码一旦被链接到内核，起**作用**与静态链接的内核代码完全**等价**；

2.3 模块的组成：

（1）模块**加载**函数（必须）
（2）模块**卸载**函数（必须）
（3）模块**许可证**声明（必须）
（4）模块参数（可选）
（5）模块导出符号（可选）
（6）模块作者等信息声明（可选）

```
#include<linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL"); //许可证声明
static int hello_init(void) //加载函数
{
    printk(KERN_ALERT "Hello, world enter\n");
    return 0;
}
static void hello_exit(void) //卸载函数
{
    printk(KERN_ALERT "hello world exit\n");
}
```

3. Hello world 模块

3.1 为什么用 printk

Printk 函数是在**内核中定义**的，功能作用与 C 库中的 printf 函数类似。**内核运行**时**不能依赖 C 库**，模块**连接内核后**可以**访问内核的公用符号**（函数、变量等）。

3.2 加载、卸载函数

__init 标识的函数，在初始化调用后，即可**释放**内存。

__exit 标识的函数在卸载时使用，若模块直接**编译进内核**或**不允许模块卸载**，则直接**丢弃**该函数。

以 module_init（）、module_exit（）的形式表示括号内的函数是对应的加载/卸载函数。

3.3 Makefile

make 指令根据 Makefile 文件规则进行编译。

多文件编译：obj-m:=hello.o

hello-objs:=file1.o file2.o

```
KERNELDIR = /usr/src/linux          /*内核源代码路径*/
PWD := $(shell pwd)                 /*当前工作路径*/
#INSTALLDIR =
/home/tekkaman/working/rootfs/lib/modules
/*内核交叉编译器路径*/
CC      = $(CROSS_COMPILE)gcc
obj-m := hello.o                    /*最终生成 hello.ko 模块*/
/*-C 进入内核源代码目录，找到顶层的 Mikefile*/
/*M 返回当前目录执行 Mikefile 文件*/
/*执行 make modules 命令，即生成模块 hello.ko*/
moudles:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
/*将生成的模块文件 hello.ko 拷贝至目标路径下*/
modules_install:
    cp hello.ko $(INSTALLDIR)
/*删除已经生成的文件*/
clean:
    rm -rf *.o *~ core .depend .*.cmd *.ko
*.mod.c .tmp_versions
```

3.4 模块加载

（1）模块直接编译进内核，随 Linux 启动时加载。

（2）内核运行时动态加载，insmod 加载，rmmod 卸载。

3.5 模块参数

insmod 时传参，否则使用默认值。

模块参数声明：

module_param(参数名，参数类型，读写权限)；

4. 重要的数据结构

4.1 struct file 与 inode

file：它代表一个打开的文件，由内核在**调用 open 时创建**，并传递给在该文件上进行操作的所有函数，直到最后的 close 函数被调用，在文件的**所有实例都关闭**时，内核**释放**这个数据结构。

file 与 FILE 无关：FILE 处于用户空间，由 C 库定义，不会出现在内核代码中。

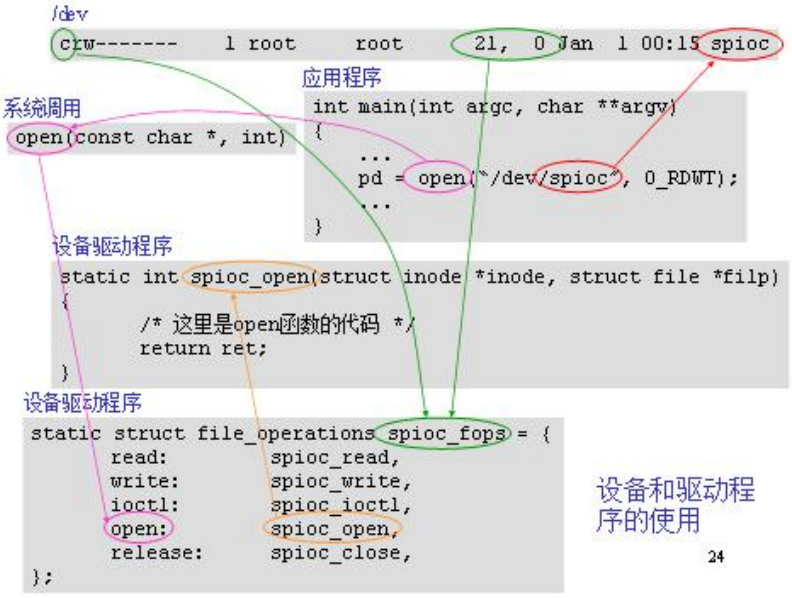
file 与 inode 不同：inode 表示文件（未打开），file 表示文件描述符。对于单个文件，可能有多个表示打开的文件描述符，但都指向同一个 inode。

```
struct     inode{
    dev_t     i_rdev;          /*对于代表设备文件的 inode，该字段包含真正的设备号*/
    struct cdev     *i_cdev;  /*该字段指向字符设备*/
}
```

4.2 file_operations 结构

file_orations 关联设备号及在其设备上的操作。

file_operations 结构中的每一个**字段**都必须指向驱动程序中实现**特定操作的函数**，对于**不支持的操作**，对应的字段可**置为 NULL** 值。对于各个函数而言，如果对应字段被赋予为 NULL 指针，那么内核的具体处理行为是不尽相同的。



4.3 cdev 结构

字符设备以 cdev 表示。

cdev 的设置：

```
cdev_init();
cdev.owner=THIS_MODULE
cdev_add();    /*设置好 cdev 后，通知内核*/
cdev_del();    /*移除一个字符设备*/
```

5. 字符设备

5.1 设备号

设备号由 12 位主设备号和 20 位次设备号组成。主设备号，表示驱动，同一类设备一般使用同一主设备号。次设备号表示设备。

设备号结构体:dev_t dev;

MAJOR(dev_t dev);

MINOR(dev_t dev);

MKDEV(major,minor);

分配设备号：推荐使用动态分配，静态分配可能会出现冲突。动态分配的缺点是分配的设备号可能不一样，无法预先创建设备节点，但脚本可以通过 awk 获得/proc/devices 信息，详情可见 scull_load 脚本。

动态分配：alloc_chrdev_region();

静态分配：register_chrdev_region();

动、静分配：register_chrdev(); /*int 为 0 动态，否则静态*/

释放设备号：

unregister_chrdev_region();

注意，若申请设备号失败，则需要释放已经申请成功的设备号。

__register_chrdev_region() 主要执行以下步骤：

1. 分配一个新的 char_device_struct 结构，并用 0 填充。

2. 如果申请的设备编号范围的主设备号为 0，那么表示设备驱动程序请求动态分配一个主设备号。动态分配主设备号的原则是从散列表的最后一个桶向前寻找，那个桶是空的，主设备号就是相应散列桶的序号。所以动态分配的主设备号总是小于 256，如果每个桶都有字符设备编号了，那动态分配就会失败。

3. 根据参数设置 char_device_struct 结构中的初始设备号，范围大小及设备驱动名称。

4. 计算出主设备号所对应的散列桶，为新的 char_device_struct 结构寻找正确的位置。同时，如果设备编号范围有重复的话，则出错返回。

5. 将新的 char_device_struct 结构插入散列表中，并返回 char_device_struct 结构的地址。

5.2 open()与 release()

open()往往是一个设备操作的发起者,而 release()往往是一个设备操作的终结者。

open 作用：

1).**检查**设备特定的**错误**(如设备未就绪);

2).**对设备实现初始化**;

3).更新 f_op 指针.这一点很实用,比如带有子系统的驱动模型,一般有系统默认的操作集,如果系统默认的操作集满足不了我们的需求时,我们可以**通过更新 f_op 指针指向我们具体的操作集**,实现了操作集的重载.其中 LCD 子系统就是这么干的;

4).**有必要时更新 filp->private_data**.当多个设备共享一套驱动时,为后续读写数据交

互,可以借助这个"桥梁".

release()完成与 open()相反的工作.释放 open()向内核申请的所有资源。

5.3 write()与 read()

通过 copy_to_user 与 copy_from_user 实现读写操作。

5.4 ioctl()

用户程序使用 ioctl 系统调用来**控制设备**。用户程序只是通过**命令码**告诉驱动程序想**做什么**，至于怎么解释这些命令和怎么实现这些命令，这都是驱动程序要做的事情。

5.5llseek()

修改当前读写位置。

6. 并发和竞态

并发与竞态：多个进程对共享资源并发访问，从而产生了竞态。

临界区：访问共享资源的代码段。

通过互斥操作来解决竞态问题：中断屏蔽、原子操作、自旋锁、信号量等。中断屏蔽很少单独被使用，**原子操作**只能针对**整数**进行，因此自旋锁和信号量应用最为广泛。

自旋锁是忙等锁，适用临界区短、无阻塞（阻塞会陷入死锁）。

信号量是睡眠锁，主要开销在于上下文切换，适用于临界区大、允许阻塞。

6.1 中断屏蔽

Linux 内核的进程调度等操作都依赖中断来实现。

关中断——访问临界区——开中断

不推荐这么做，关中断使得所有中断无法处理。

6.2 原子操作

原子的操作指的就是在执行过程中**不会被别的代码所中断**的操作。

使用原子变量使设备只能被一个进程打开？

```
static atomic_t xxx_available = ATOMIC_INIT(1);/*定义原子变量*/
static int xxx_open()
{
  ..........
  if(!atomic_dec_and_test(&xxx_available )){
     atomic_inc(&xxx_available);
     return BUSY;//已经打开
  }
  .......
  return 0;//成功
}
int release()
{
  atomic_inc(&xxx_available);//释放设备
}
```

原子变量初始化为 1，若没有进程打开则“原子变量-1=0”；atomic_dec_and_test()函数为真，取反为假不执行 if 语句。若有进程打开，则变量为-1，执行 if 语句。

release 函数中，atmoic 加一，变量值为 1 。

6.3 自旋锁（适用于自旋锁占用**时间短**，临界区**不可阻塞**）

自旋锁实际上是**忙等锁**，当锁不可用时，CPU 一直循环执行 "测试并设置" 直到可以取得该锁，CPU 在等待自旋锁时不做任何有用的工作，仅仅是等待。

只有在**占用锁时间极短**的情况下，使用自旋锁才是合理的。当临界区很大或有共享设备的时候，需要较长时间占用锁，使用自旋锁会降低系统的性能。

自旋锁可能导致系统死锁。引发这个问题最常见的情况是递归使用一个自旋锁，即如果一个已经拥有某个自旋锁的 CPU 想第 2 次获得这个锁，则该 CPU 死锁。此外，如果进程获得自旋锁后再阻塞，也有可能导致死锁的发生。copy_from_user()、copy_to_user() 和 kmalloc() 等函数都有可能引起阻塞，因此在自旋锁的占用期间不能调用这些函数。

```
int xxx_count = 0;    /*定义文件打开次数*/

static int xxx_open (struct inode *inode, struct file *filp)
{
     ...
     spin_lock (&xxx_lock);  /*获取自旋锁*/

     if (xxx_count){     /*获得自旋锁,但已有别的进程打开设备*/
          spin_unlock (&xxx_lock);    /*设备忙,释放自旋锁返回*/
          reurn (-EBUSY);
     }
     xxx_count++;         /*没有别的进程使用设备,增加使用计数*/
     spin_unlock (&xxx_lock);    /*已经成功打开设备,释放自旋锁*/
     ...
     return (0);         /*打开成功*/
}
static int xxx_release (struct inode *inode, struct file *filp)
{
    ...
    spin_lock (&xxx_lock);    /*获得自旋锁*/
    xxx_count--;        /*减少使用计数*/
    spin_unlock (&xxx_lock);
    return (0);
}
```

自旋锁保护的是 count 变量。

6.4 信号量

信号量是一种**睡眠锁**。如果有一个任务试图获得一个已被持有的信号量时，信号量会将其**推入等待队列，然后让其睡眠。**这时处理器获得自由去执行其它代码。当持有信号量的进程将信号量释放后，在等待队列中的一个任务将被唤醒，从而便可以获得这个信号量。

7. 阻塞与非阻塞

同步和异步关注的是**消息通信机制。**

同步，就是在发出一个功能调用时，在**没有得到结果之前，该调用就不返回。**

异步过程调用发出后，调用者**不能立刻得到结果。**实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。

阻塞与非阻塞关注的是**程序在等待调用结果（消息，返回值）时的状态**。

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。

非阻塞和阻塞的概念相对应，指在不能**立刻得到结果之前**，该函数不会阻塞当前线程，而会**立刻返回**。

7.1 等待队列（阻塞）

等待事件：wait_event_interruptible(queue,condition);

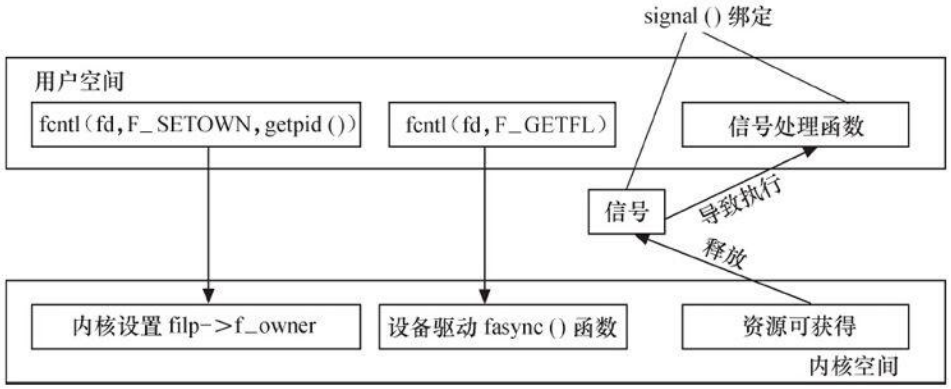
唤醒等待队列：wake_up_interruptible (wait_queue_head_t *q);

例：想象一下如下代码，当进程读设备时进入睡眠而在其他人写设备时被唤醒的情景?

```
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;
ssize_t sleepy_read (struct file *filp, char __user *buf, size_t count, loff_t *pos)
{
        printk(KERN_DEBUG "process %i (%s) going to sleep\n",
                current->pid, current->comm);
        wait_event_interruptible(wq, flag != 0);
        flag = 0;
        printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
        return 0; /* EOF */
}
ssize_t sleepy_write (struct file *filp, const char __user *buf, size_t count, loff_t *pos)
{
        printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
                current->pid, current->comm);
        flag = 1;
        wake_up_interruptible(&wq);
        return count; /* succeed, to avoid retrial */
}
```

7.2 异步通知（非阻塞）

异步通知的意思是：一旦**设备就绪，则主动通知应用程序**，这样应用程序根本就不需要查询设备状态，这一点非常类似于硬件上“中断”概念，比较准确的名称是“信号驱动的异步I/O”。



应用程序端捕获：信号一个信号被捕获的意思是当一个信号到达时有相应的代码处理它。

（1）首先**指定一个进程作为文件的属主**。通过使用 **fcntl** 系统调用执行 F_SETOWN 命令时，属主进程的 ID 号就会保存在 filp->f_owner 中，目的是为了让内核知道应该通知哪个进程。fcntl(STDIN_FILENO, F_SETOWN, getpid());

（2）在设备中**设置 FASYNC 标志**。通过 fcntl 调用的 F_SETFL 来完成。

oflags=fcntl(STDIN_FILENO,F_GETFL)；

（3）void (*single(int signum,void (*handler)(int))) (int);

第一个参数为指定信号的值；第二个参数为指定针对前面信号值的处理函数；

驱动程序释放信号：

（1）支持 F_SETOWN 命令，设置 filp->f_owner 为对应的进程。

（2）支持 F_SETEL 命令，每当 FASYNC 标志改变时，驱动程序中的 fasync（）得以执行。

处理标志变更函数 int fasync_helper()

（3）设备资源可获得时，调用 kill_fasync()激发相应信号。

释放信号函数 void kill_fasync()

在文件关闭是，release（）函数中，将文件从异步通知列表中删除，xxx_fasync(-1,filp,0)

8. 字符设备、块设备、网络设备

字符设备：提供连续的数据流，应用程序可以**顺序读取**，通常不支持随机存取。相反，此类设备支持按字节/字符来读写数据。举例来说，键盘、串口、调制解调器都是典型的字符设备。

块设备：应用程序可以**随机访问**设备数据，程序可自行确定读取数据的位置。硬盘、软盘、CD-ROM 驱动器和闪存都是典型的块设备，应用程序可以寻址磁盘上的任何位置，并由此读取数据。此外，数据的读写只能**以块(通常是 512B)的倍数进行**。与字符设备不同，块设备并不支持基于字符的寻址。

网络设备：网络设备是特殊设备的驱动，它**负责接收和发送帧数据**，可能是物理帧，也可能是 ip 数据包，这些特性都有网络驱动决定。它**并不存在于/dev 下面，**所以与一般的设备不同。网络设备是一个 net_device 结构，并**通过 register_netdev 注册到系统里，**最后通过 ifconfig -a 的命令就能看到。