内核模块之间的交互，export\_symbol

随内核启动的模块输出，还是手工动态加载输出有区别

1. EXPORT\_SYMBOL标签内定义的函数或者符号对全部内核代码公开，不用修改内核代

码就可以在您的内核模块中直接调用，即使用EXPORT\_SYMBOL可以将一个函数以符号的方式导出给其他模块使用

使用时注意事项：

在使用EXPORT\_SYMBOL 的.c文件中 需要 #include <linux/module.h> 文件。

先写函数

func\_a ()

{

…

}

再使用EXPORT\_SYMBOL

EXPORT\_SYMBOL(func\_a);

而要调用func\_a的模块在加载时，会先确认该模块调用的各内核函数是否已export（参看\_\_find\_symbol() kernel/module.c）

一个模块mod1中定义一个函数func1；在另外一个模块mod2中定义一个函数func2，func2调用func1。

在模块mod1中，EXPORT\_SYMBOL(func1);

在模块mod2中，extern int func1();

就可以在mod2中调用func1了。

具体代码见附1

1. volatile与static变量

当要求使用volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。

有时候变量被频繁读取时，会把变量存放在cache中，使CPU不用去内存读取数据提高频率，但是如果该变量改变很快，内存中已经更新，但是cache中并没有更新的话，而要实时的读取该数据的话，把该数据定义为volatile型，则像上面所说的那样，CPU直接从内存读取该数据

　static 声明的变量在C语言中有两方面的特征：

　　1)、变量会被放在程序的全局存储区中，这样可以在下一次调用的时候还可以保持原来的赋值。这一点是它与堆栈变量和堆变量的区别。

变量用static告知编译器，自己仅仅在变量的作用范围内可见。这一点是它与全局变量的区别。

若全局变量仅在单个C文件中访问，则可以将这个变量修改为静态全局变量，以降低模块间的耦合度；

若全局变量仅由单个函数访问，则可以将这个变量改为该函数的静态局部变量，以降低模块间的耦合度；

设计和使用访问动态全局变量、静态全局变量、静态局部变量的函数时，需要考虑重入问题；

静态变量放在程序的全局数据区，而不是在堆栈中分配，所以不可能导致堆栈溢出

1. unsigned char为无符号字符类型

printf()函数只识别第一个参数,即格式字符串,对于后一个参数,如你不指定,则当作32位的int来处理. 此时进行符号扩展,变成32位。当作signed char打印的时候,输出时转化为unsigned 故输出8位

附1

mod1.c

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

static int func1(void)

{

printk("In Func: %s...\n",\_\_func\_\_);

return 0;

}

EXPORT\_SYMBOL(func1);

static int \_\_init hello\_init(void)

{

printk("Module 1,Init!\n");

return 0;

}

static void \_\_exit hello\_exit(void)

{

printk("Module 1,Exit!\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

#############################################################

mod2.c

#include<linux/init.h>

#include<linux/kernel.h>

#include<linux/module.h>

static int func2(void)

{

extern int func1(void);

func1();

printk("In Func: %s...\n",\_\_func\_\_);

return 0;

}

static int \_\_init hello\_init(void)

{

printk("Module 2,Init!\n");

func2();

return 0;

}

static void \_\_exit hello\_exit(void)

{

printk("Module 2,Exit!\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

################################################################

Makefile

ifneq ($(KERNELRELEASE),)

obj-m := mod1.o

else

KDIR := /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

default:

$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:

rm -rf Module.symvers \*.ko \*.o \*.mod.c .\*.cmd .tmp\_versions

endif

################################################################

第二个 一样的mod2.o

#insmod ./mod1.ko

#insmod ./mod2.ko

#rmmod mod2

#rmmod mod1

Jan 11 11:59:17 wangyao-desktop kernel: [ 9886.801010] Module 2,Exit!

Jan 11 11:59:21 wangyao-desktop kernel: [ 9891.450214] Module 1,Exit!

Jan 11 12:05:29 wangyao-desktop kernel: [10258.385014] Module 1,Init!

Jan 11 12:05:38 wangyao-desktop kernel: [10267.465923] Module 2,Init!

Jan 11 12:05:38 wangyao-desktop kernel: [10267.465928] In Func: func1...

Jan 11 12:05:38 wangyao-desktop kernel: [10267.465930] In Func: func2...

Jan 11 12:05:50 wangyao-desktop kernel: [10280.091755] Module 2,Exit!

Jan 11 12:05:57 wangyao-desktop kernel: [10287.332596] Module 1,Exit!

可见，在mod2中的func2函数成功的调用了mod1中的func1函数。

注意：

在编译mod2的时候，出现一个WARNING：

root@wangyao-desktop:~/modules/export\_symbol/mod2# make

make -C /lib/modules/2.6.22-14-generic/build SUBDIRS=/root/modules/export\_symbol/mod2 modules

make[1]: Entering directory `/usr/src/linux-headers-2.6.22-14-generic'

Building modules, stage 2.

MODPOST 1 modules

WARNING: "func1" [/root/modules/export\_symbol/mod2/mod2.ko] undefined!

make[1]: Leaving directory `/usr/src/linux-headers-2.6.22-14-generic'

这主要是因为在编译连接的时候还没有和内核打交道，当然找不到symbol了，但是由于你生成的是一个内核模块，所以LD不提示error，而是给出一个warning，寄希望于在insmod的时候，内核能够把这个symbol连接上。

有时候一个工程包含很多文件，不可能把所有的编译规则写在一个Makefile文件里面

比如一个文件夹下有两个子文件夹，里面各有各的文件和Makefile

--------------------gupeng文件夹---------------------------------------------------

-----------------------dir1----------------- 顶层Makefile -----------------dir2-----------------------

File1 Makefile File2 Makefile

顶层Makefile 的写法：

subsystem:

cd dir1 && $(MAKE)

cd dir2 && $(MAKE)

就这么简单的3行话，分别进入dir1，dir2执行里面的第二层Makefile