

SIXUENET

To Learn Without Thinking Is To Be Useless, To Think Without Learning Is To Lose
(<http://www.mysixue.com/>)

KERNEL MODULE

📅 March 27, 2019 (<Http://Www.Mysixue.Com/?P=58>) 👤 JL
(<Http://Www.Mysixue.Com/?Author=1>) 🔔

1. Introduction to document structure

First, we will give a macro introduction to the kernel module in Chapter 2 , and give a demo, so that we can quickly get started and experience how to compile and use the kernel module .

Next will be the first 3 doing chapter details the kernel module for more detailed analysis , so that we can understand in depth kernel module , and write a professional kernel module . This chapter requires repeated reading comprehension . Could over time or experience When it comes to specific items , you need to take them out and re-read them .

Finally, the first 4 chapters is driven design guide specification , it is an outline nature , remind us of the main considerations when writing kernel modules . When we make a specific project , you can query this chapter to obtain a quick guide , if you fast If you forget the details of the guidelines , you need to re-understand the content of Chapter 3 .

2. Introduction to the kernel module

2.1 What is a kernel module

The boot sequence of Linux on the ARM board is generally boot->kernel image->filesystem.

Kernel Image is a file , its name is commonly called zImage or uImage.

We are how to compile a kernel image of it ? General steps : First, get the kernel source code ; then run make menuconfig command to configure the source code , the so-called configuration is the kernel function to customize , such as the need LCD function , put The LCD- related code is configured as Y. If the audio playback function is not required , the audio-related code is configured as N, and there is another type that can be configured as M; after the configuration is completed , execute the make command , and we will get the kernel image file we need .

Execution make the process of , all configured Y codes are compiled and linked into the kernel image which , configured as a Y more code , the generated image greater ; all configured as N code will not compile ; all configured The M option will generate the corresponding .ko file , **this .ko file is the kernel module** .

Kernel modules and kernel image position is such , kernel image which contains a number of fixed function of the kernel , such as LCD display , touch and the like ; kernel module to extend functionality of the kernel , such as a USB Bluetooth device of lany.ko file . kernel modules can be dynamically loaded into the kernel , such as the lany.ko loaded into the kernel , the kernel has a Bluetooth function ; the same kernel module can also be dynamically unloaded from the kernel , such as unloading lany.ko, the kernel will lose the Bluetooth function .

So what are the benefits of kernel modules and this dynamic loading mechanism of modules ?

- ❶❷❸❹❺❻❼❽❾❿ all , the module itself is not compiled into the kernel image, thus controlling the size of the kernel image
- Secondly , once the module is loaded , it and the kernel image other portions exactly the same , so that the extended functionality of the kernel
- In commissioning the drive time , the mechanism of dynamic loading will be very easy , you just need to recompile the kernel module , and reload it can be ; without the need to repeat the " compiled kernel image, burn kernel image, reboot Kernel Image ." This A long process .

2.2 Quickly experience the kernel module

In addition to the way to compile the kernel module by configuring M when making menuconfig mentioned in section 2.1 (in this way , the module code is placed in a certain directory of the kernel source code) , there is another way to compile the module .

We can write the code in a separate module , not including it in the kernel source tree , just in time compiler module , specify the kernel tree path can be . In this way, when the kernel is already running on a machine will be very Convenient .

The Demo in this section uses the latter method .

Find a machine running a Linux system , such as a PC running ubuntu , or a Raspberry Pi running debian , and do the following

- Obtain the module code
git clone https://gitlab.com/study-linux/building_running_modules.git
- Compile the module
cd building_running_modules && make
- Load module
sudo insmod HelloWorld.ko

After loading, run the dmesg command , you will see the following message at the end

```

4261.636016] Hello Josh
4261.637804] array element input from user: 0
4261.638651] int_array[0]: 1
4261.639359] int_array[1]: 2
4261.640076] int_array[2]: 3
4261.640727] int_array[3]: 4
4261.641443] int_array[4]: 5
4261.642378] int_array[5]: 6

```

- ❶❷❸❹❺❻❼❽❾❿ module
sudo rmmod HelloWorld

After uninstalling, run the dmesg command , and you will see the following message at the end :

```

[ 4456.824315] Goodbye Josh

```

3. Detailed explanation of kernel modules

3.1 Comparison of kernel modules and applications

This section describes in more detail what is the kernel module , mainly the kernel modules and applications do the right ratio .

First explain design ideas under each heading , the title role is given outline , emphasize important points . When we finished and detailed understanding of this article , over a period of time , some details may forget . When I wanted to recall these details time , the need to read through the article from beginning to end in it ? may be possible , but a waste of time , and we've been careful to understand this article , just to have an outline , you can think of it inside details .

Therefore, each of the following section heading , represents the difference between the kernel modules and applications , open navigation bar , reading these titles , can quickly understand or recall something . Some title a little longer , because we want to be more clearly express standard meaning of the title , the navigation bar to the right and more drag one o'clock , so you can see all the titles .

Kernel space and user space

Here briefly kernel space and user space , a more comprehensive explanation would put Linux kernel subsystem on process scheduling and memory management articles in . Kernel code and user space code runs in a virtual address , a 32 -bit CPU, virtual There are a total of 4G addresses . The kernel code uses 3G ~ 4G space , called kernel space ; applications use 0 ~ 3G space , called user space .

In addition to the address space is different , the priority level is also different . The code in the kernel space runs at the highest level , in which all operations can be performed ; while the application runs at the lowest level , in this level , the CPU controls the hardware Direct access to memory and unauthorized access to memory .

The kernel module is part of the kernel code , so it also runs in the kernel space .

The application program runs in the user space . Whenever the application program executes a kernel system call or is suspended by a hardware interrupt , it enters the kernel space . The kernel code that implements the system call function runs in the process context, and it performs operations on behalf of the user process . the hardware interrupts kernel code and the process is asynchronous , independent of a particular process .

Generally , a kernel module (also called the driver), need to process these two tasks , performed as part of some function modules as system calls , the other function is responsible for interrupt handling .

Based on the above concepts , we can also understand that the two processes in the user layer cannot exchange data , that is , they cannot communicate . Because each process has a virtual address space of 0 ~ 3G , there is no way for one process to give another one. The process passes an address so that another process can obtain data from this address . Because the same virtual address will be mapped to different physical addresses in different processes , there is no way to exchange data . If two processes want to communicate , they need to pass through the kernel. , Because different processes see the same kernel space , the same kernel space virtual address will be mapped to the same block of memory ; process 1 enters the kernel space through a system call , puts a data in an address in the kernel space , and then the process 2 also enters kernel space , away from the data at this address , to complete the communication .

For example, to imagine a process of each city, each city's metro imagine virtual address for the process. Regulations for Common city subway name only from 1-10 in the capital subway named from 10 - 15. Then you are in city A's Metro 2 put a bag on a seat number line, so that the city B people to the subway 2 take this bag on the seat number line, so he went to his 2 number online to look, obviously this is not available. take the package to him, you need to go 10 somewhere down the line number of the package, and then told him to go 10 line pick up. of course, only to enter the capital, to see the 10 line.

So there may be multiple processes to call our kernel module. The kernel code can get the current process by accessing the global item `current` < defined in `asm.current.h` >. `current` is a pointer of `struct task_struct` type, through `current`, we can Get the detailed information of the process, such as the command name `current->comm`, and the process ID `current->pid`.

Kernel modules are similar to event-driven methods, and applications are not all

What is called an event-type drive? Such as a lazy child, what does not move up and sit in the morning, he went to his mother to let him brush your teeth brush your teeth, let him eat his breakfast and had breakfast, he will not take the initiative to do anything, and others let him do what he did, this would be called an event-type drive. design codes are also often encounter this line of thinking, the most common of such a message handler, when someone sends to its message queue inside when a message, and he took the news, determine what is news, then post-processing of the message. this procedure, like with the child, they have the ability to do certain things, but do not take the initiative, but waiting for the event to drive It does.

The kernel module is like this. When it is loaded, it will tell the kernel: "I'm here, I can handle XXX things", and then it won't move. When the kernel needs a certain function, it will Call the interface it provides; when uninstalling, it will tell the kernel: "I'm leaving, don't look for me for XXX things in the future."

Most small-scale and medium-scale applications, not event-driven, they all start to finish complete execution of the code, and then quit. Some applications are also designed to be event-driven, but it is the core module of any course there is a major difference: the application when you exit, you can not manage cleanup work release or other resources; **but exit function kernel modules must be carefully undo everything done initialization function**, otherwise the system reset before the start of these things will be left In the kernel.

The kernel module can only call functions exported by the kernel

As programmers, we know that an application can call functions that it does not define, because the connection process can find the program in a library. For example, the `printf` function defined in `libc` can be called by any application.

But the kernel modules are linked only to the kernel, it can only call those functions kernel exported, without any library can be linked in. For example, `printk` is a core function of export, when we write kernel code, you can only use `printk`, Instead of using `printf`.

To see which functions are exported by the kernel, you can get it through the `cat /proc/kallsyms` command, or in the kernel source code, `grep EXPORT_SYMBOL`

Kernel programming must always consider concurrency issues

Another difference between kernel programming and programming of applications , kernel programming must always remember , even the simplest kernel module , should consider the issue of concurrency .

Removal of multi-threaded applications , most applications are executed sequentially , need not be concerned because the occurrence of a number of other things will change their operating environment . Kernel code , but not in such a simple world running .

There are several reasons why kernel programming must consider concurrency issues . First , there may be multiple concurrent processes using our driver at the same time ; second , most devices can interrupt the CPU, the interrupt handler is asynchronous, and the driver may be ready to variable a assigned 1 Shi , interrupt generated , the interrupt handler system calls the driver's requested that a assigned to 4; as well as on multi-processor systems , may also be more than one CPU to run our drivers ; and finally , The kernel code in 2.6 is already preemptible, which means that even on a single processor, there are similar concurrency problems in a multi-processor system .

Therefore , the Linux kernel code (including driver code) must be able to run in multiple contexts at the same time , the kernel data structure needs to be carefully designed to ensure that multiple processes are executed separately , and the code that accesses the shared data must also avoid destroying the shared data .

To write correct kernel code , good concurrency management is a must . We need more comprehensive knowledge to do this well . We will use a separate article to discuss concurrency issues and concurrency management in the kernel. Primitive .

The kernel space stack is very small , so large data should be allocated dynamically

Briefly about the heap and stack difference , local variables stored in the stack in , parameters of the function call , when a function calls another function inside the context of this function , is stored in the stack in . Global and static variables are placed Static storage area (their addresses are determined during compilation) .The memory from malloc or new is in the heap and needs to be managed and cleared by the programmer .

Applications in virtual memory layout , and there was a lot of stack space . Kernel stack space is very small , probably only with a 4096 -byte pages as small , our kernel module code sharing this with the entire kernel space code Stack , so you need to pay attention to the use of the stack . Generally large data structures , it is best to use dynamic allocation .

APIs starting with __ in the kernel , call carefully

Kernel API export function or the kernel , there are many two underscores (__) as a function of the prefix , the function of this name is usually some underlying components , should be used with caution , or peril .

In general , __ the beginning of the function , there will be one without __ prefix function package it , we'd better use these functions .

The kernel code cannot implement floating point operations

The kernel code cannot implement floating-point operations . If floating-point support is turned on , on some architectures , it is necessary to save and restore the floating-point processor state when entering and exiting the kernel space . This additional overhead has no value , and the kernel code also No floating point operations are required .

So printf can print floating point numbers , but printk does not support floating point numbers .

Continue to update the content of this section

Kernel programming is different from user space programming in many aspects . In the later learning process , we will continue to add these differences in this section .

3.2 Compilation and loading of kernel modules

Compile module

At the beginning of this article, I quickly experienced how to compile a kernel module . You only need to type a make command , and then you can get the result . But in fact, the things behind it are more than that simple . This section will introduce some details of module compilation in detail .

Before starting to compile the module , you need to do some preparatory work .














First of all , check whether the compiler version is correct , whether the module tools and other necessary tools are complete , the kernel document directory Documentation/Changes file lists the required tool versions . Do not use old tools , and do not use newer versions of tools .

Secondly , the need to prepare the kernel tree and construct the kernel tree .

There are two cases to discuss this kernel tree :

The first situation is when we do board-driven development , for example, we have a board of raspberry pie , to compile a kernel runs on the board , we will send the official from raspberry github repository clone kernel source code into a A directory , enter the directory to configure and compile the kernel . This source code is the kernel tree , and the operation of compilation is to construct the kernel tree .

The second case is that we already have a machine running a Linux system , such as a PC running ubuntu . If the machine does not have a kernel tree installed , we need to install and construct the kernel tree . The specific steps are as follows :

- Run the command `uname -r` , the result is as follows: 3.2.0-48-generic
- Check `/lib/modules` whether a directory corresponding to the directory , the form : `/lib/modules/3.2.0-48-generic` . If present , demonstrate the number of cores has been installed , `/lib/modules/3.2.0-48-generic/Build` file is a link , pointing to kernel source tree directory is located ; if not , you will need to be installed and configured according to the following steps the number of cores .
- Run the command `sudo apt-cache search linux-source` to check the downloadable source package
- Choose a corresponding source package to install , like : `sudo apt-get install linux-source-3.2.0`
- Once downloaded , will be in `/usr/src` appear at a `linux-source-3.2.0.tar.bz2` archive
-    the compressed package : `tar jxvf linux-source- 3.2.0 .tar.bz2`
-    decompression is complete , a new directory will appear : `/usr/src/ linux-source- 3.2.0`
-    the directory : `make oldconfig && make && make bzImage && make modules && make modules_install`
-     execution is complete , you will see the corresponding directory under `/lib/modules`

After the preparations are complete , you can start to compile your own kernel modules .

Suppose you have written a very simple hello.c and want to compile it into a module

Just create a new Makefile file in the directory where hello.c is located , and add a sentence of obj-m := hello.o

Then compile command the make - C ~ / Kernel-2.6 M = ` pwd ` modules

The above-described first change directory command to -C location option established (assuming kernel source tree in ~ / Kernel-2.6) , which holds the kernel directory top Makefile file , M = option allows top Makefile in the configuration modules before the target , back to block The directory where the source code is located , and then the modules target points to the module set in the obj - m variable .

Familiar with the Makefile people may have noticed that the kernel module Makefile is very simple , with the Makefile common form is not the same , this is because the kernel build system deal with the remaining issues . Kernel configuration system is very complicated , if you need to know the whole picture , you can read Documentation Files in the /kbuild directory .

obj-m = hello.o show that there is a need from the object module file hello.o configuration , the name of the module is Hello. Ko

If the name of a module we want to construct is module.ko, and it is generated from two source files (file1.c, file2.c), the correct Makefile can be written as follows :

```
obj-m := module.o
```

```
module-objs := file1.o file2.o
```

It is annoying to have to type the above long list of make commands every time . We can design the Makefile a little bit . The first purpose is to compile the module with just one make command ; the second purpose is to no matter where the module code is placed. , can be compiled , the so-called " no matter where on " , refers to whether you placed source tree directory , or on the source tree outside , can be compiled . the Makefile : https://gitlab.com/study-linux/building_running_modules/blob/master/Makefile (https://gitlab.com/study-linux/building_running_modules/blob/master/Makefile)

```
# If KERNELRELEASE is defined, we've been invoked from the
```

```
# kernel build system and can use its language.
```

```
ifneq ($(KERNELRELEASE),)
```

```
    obj-m := HelloWorld.o
```

```
# Otherwise we were called directly from the command
```

```
# line; invoke the kernel build system.
```

```
else
```

```
    KERNELDIR? = /Lib/modules/$(shell uname -r)/build
```

```
    PWD := $(shell pwd)
```

```
default :
```

```
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
```

```
endif
```

```
clean :
```

```
    rm -f *.o *.ko
```

distclean :

```
rm -f *.o *.ko *.mod.c modules.* Module.*
```

If we put the module code in the number of cores directory , the number of cores perform the root directory make command , the `KERNELRELEASE` is not empty , execution if part generating module ; If we put the code in the module number of cores directory outside , the Makefile is read 2 times , the first time we knock on the make after the command , `KERNELRELEASE` is empty , performs else part , else some settings `KERNELDIR` , `PWD`, and then perform default under the `$ (MAKE)` command (Makefile inThe make command is parameterized as `$(MAKE)`), and then the make command will find the kernel tree , and then call the Makefile for the second time . This time , `KERNELRELEASE` is not empty , and the if part is executed to generate the module . `KERNELRELEASE` is in the kernel number Initialized in the top-level Makefile .

Load / unload modules

After the module is successfully constructed , the next step is to load the module .

`insmod` can be used to complete this work . The `insmod` program is similar to `ld` . It loads the code and data of the module into the kernel , and then uses the kernel symbol table to resolve any unresolved matches in the module . The difference from the linker is that `insmod` The kernel's disk file will not be modified , only the copy of the inner seed will be modified , which means that after restarting , you need to `insmod` again .

`insmod` can accept some command line options (man `insmod`), and can assign values to integer and string variables in the module before the module is linked to the kernel . That is to pass parameters to the module , we will discuss how to write module code in detail later Make it accept parameters .

`insmod` depends on the definition in `kernel / module.c` a system call , function `sys_init_module` to allocate kernel memory module , kernel code and only system call name with a front `sys_` prefix . If you are interested in learning `insmod` details , you can go Find it online , there is not much discussion here .

`rmmmod` is used to remove modules from the kernel . Note that if the kernel believes that the module is still in use , or the kernel is configured to prohibit removal of the module , the module cannot be removed .

The `modprobe` tool is similar to `insmod` and can also be used to load modules . The difference is that `modprobe` will consider whether the module to be loaded refers to some symbols that do not exist in the current kernel . If so , `modprobe` will look for these in the current module search path. Symbol of other modules , and load these modules at the same time . That is to say, `modprobe` will handle the dependencies between modules .

In use `modprobe` time , note the following :

`modprobe` is looking for the loaded module under `/lib/module/`uname -r`` , and `modprobe` needs a newest `modules.dep` file . The content of this `modules.dep` file is the dependency between each module and other information . This file is It is updated by the `depmod` command .

And `insmod` difference is , `modprobe` with the name of the module followed by the module , not the name of the module file , that is not need to bring `.ko` suffix .

The specific steps used by `modprobe` are as follows :

The compiled modules into `/lib / module / `uname -r`` under

Update the `modules.dep` file with the `sudo depmod` command

Then use the `sudo modprobe HelloWorld` command to load the module

lsmod lists all modules currently loaded in the kernel , which is obtained by reading the / proc / modules to obtain such information the virtual file , information about the currently loaded modules , may be in sysfs virtual file system / sys / module case Found .

3.3 Write kernel module code

After understanding the above information , we can start to automatically write the kernel module . The macroscopic idea is that there is a c file that stores the source code of the module we have written , and there is a Makefile to compile the c file , and then execute make Command to compile the kernel module , and then you can load / unload the kernel module .

In this section , we will explain explain the c file how to write , the kernel is a specific environment , the need for it and the kernel module interface code has some special requirements .

Header files and MODULE_XXX

Most of the kernel code must include a considerable number of header files in order to obtain the definitions of functions, data types and variables . There are several header files specifically for modules , so they must appear in every loadable module . Therefore , All modules-generation code contains the following two lines of code :

```
#include <linux/module.h>
#include <linux/init.h>
```

In addition , the module should specify the license used by the code . So we need to include the MODULE_LICENSES line :

```
MODULE_LICENSES( " GPL " );
```

The licenses recognized by the kernel include : " GPL " , " GPL v2 " , " GPL and additional rights " , " Dual BSD/GPL " , " Dual MPL/GPL " , " Proprietary (proprietary) " . If a module does not have display marked as described above permits identification kernel , it will be assumed to be proprietary , when the module is loaded , the kernel receive the contaminated (kernel tainted) warnings .

In addition to the above-described LICENSES, the module may also selectively contain descriptive defined as follows :

MODULE_AUTHOR: describe the author of the module

MODULE_DESCRIPTION: A short description used to illustrate the purpose of the module

MODULE_VERSION: Code revision number , for rules about version strings , refer to the comments in linux/module.h

MODULE_ALIAS: the alias of the module

MODULE_DEVICE_TABLE: Tell the user space module supported devices

The above MODULE_ declaration can appear anywhere in the source file where the function thinks . The general practice is to put these declarations at the end of the file

Module loading function and error handling during initialization

Load function

The load function of the module is necessary for the kernel module code . The load function is generally as follows

:

```
static int __init initialization_function(void)
{
    /* Initialization code here */
}
```

```
module_init( initialization_function);
```

Explain a few key points in the above code snippet :

static	C language static when the modification function , hidden meaning , indicates that the function can only be used in this document
int	The return value of the load function is an integer , 0 means success , if it fails , an error code (linux/errno.h) should be returned , and the error code is a negative value , such as -ENOMEM
__init	It is a hint to the kernel , indicating that the function is only used during initialization . In the kernel , all functions identified by __init are placed in the .init.text section when linking . In addition , all __init functions are in the section .initcall.init also saves a copy of function pointers , the kernel will call these _init functions through these function pointers during initialization , and release the init section (including .init.text, .initcall.init, etc.) after the initialization is completed . after the module is loaded , the loader will load the lost function , to release the memory occupied function . This is optional , you do not have to add , but added it is recommended , in order to save memory . Note, however , function or data structure after the initialization do not want to use any (with __initdata mark) to use this mark . I smod can view the memory occupied by the module , we can do a comparison experiment .
module_init	It must be used to mark a function as a load function

Loading function is also called initialization function , we usually register with the kernel initialization function inside the module (call provided by the kernel register_xxx function) , in order to perform certain functions . Like to tell the kernel , I have XXX these functions , to achieve these The functions of the functions are YYY, and the kernel will call these functions when needed . For example , we register a character device driver in the module code, and then the module can respond to system calls such as open and read .

Most of the registered function names in the kernel code are prefixed with register_ , and grep register_ can be used to easily see which registered functions are available .

Error handling during initialization

When writing module initialization code , always keep in mind that the initialization process may fail anywhere : even the simplest memory allocation , there may be insufficient memory . Therefore, the module code must always check the return value .

If you encounter an error situation , you must first determine whether the module can continue to initialize . Generally , after a certain registration fails, you can continue to operate by reducing the function .

If some error occurs and the initialization cannot be continued , you must cancel any registration work before the error . Otherwise, the kernel will be unstable because of pointers containing some non-existent code .

Error recovery is sometimes more effective with goto , which can make the logic clearer . For example :

```
int __init my_init_function(void)
{
    int err;
```

```

/* registration takes a pointer and a name */
err = register_this(ptr1, "skull");
if (err) goto fail_this;
err = register_that(ptr2, "skull");
if (err) goto fail_that;
err = register_those(ptr3, "skull");
if (err) goto fail_those;

return 0; /* success */

fail_those: unregister_that(ptr2, "skull");
fail_that: unregister_this(ptr1, "skull");
fail_this: return err; /* propagate the error */
}

```

Another method is to call the uninstall function of the module when the initialization error occurs , and in the uninstall function, only the steps that have been successfully completed in the initialization code are rolled back . This method requires more code and CPU time , and it is in the pursuit of efficiency code It is recommended to use goto.

When the initialization and removal work involves many devices , goto is a bit of a headache to use , and when adding / deleting a certain registration job , the modification of goto needs to be very careful . At this time , we can consider calling when an error occurs The method of module unloading function , the following is an example :

```

struct something *item1;
struct somethingelse *item2;
int stuff_ok;

void my_cleanup(void)
{
    if (item1)
        release_thing( item1);
    if (item2)
        release_thing2( item2);
    if (stuff_ok)
        unregister_stuff( );
    return ;
}

int __init my_init(void)
{
    int err = -ENOMEM;

    item1 = allocate_thing( arguments);
    item2 = allocate_thing2( arguments2);
    if (!item2 || !item2)
        goto fail;
    err = register_stuff(item1, item2);
}

```

```

if (!err)
    stuff_ok = 1;
else
    goto fail;
return 0; /* success */

fail :
    my_cleanup( );
    return err;
}

```

Loading competition issues

At the same time, we need to pay attention to the so-called loading competition . Once we call the kernel registration function to register a function , some parts of the kernel may immediately use the function we just registered . In other words , the module's initialization function is still running time , the kernel may call our modules . Therefore , it must be remembered , before all the initialization operation of a function is complete , do not register this function .

For example , we want to register a camera driver . Suppose we first call the kernel function to register the driver , and then initialize some hardware states of the camera . There is a problem with this logic . You may find that the camera driver you make has Sometimes it can be used , sometimes it can't . At this time you are crazy ...

In addition, it should be noted that if we have successfully registered a certain function , but an error occurs in the following initialization code , the kernel may already be calling those functions that you registered successfully . We should try to avoid this situation , in case there was , in dealing with error conditions , require careful handling core may be operating other parts of the process , and wait for the completion of these operations .

Module uninstall function

The uninstall function of the module is also necessary for the kernel module code . The uninstall function is generally as follows :

```

static void __exit cleanup_function(void)
{
    /* Cleanup code here */
}

module_exit( cleanup_function);

```

Explain a few key points in the above code snippet :

static	C language static when the modification function , hidden meaning , indicates that the function can only be used in this document
void	The uninstall function has no return value
__exit	<p>The modified function is only used for module uninstallation and can only be called when the module is uninstalled or the system is shut down . If the module is directly embedded in the kernel , or the kernel configuration does not allow t he module to be uninstalled , the function marked as __exit will be simply Discarded .</p> <p>Therefore , if the uninstall function is called during the error handling of the initialization process , it should not be marked as __exit.</p>

module_exit	It must be used to mark a function as an uninstall function

Generally speaking , the unloading function has to perform the opposite function of the module loading function , for example :

If the loading function registers XXX, the unloading function needs to unregister XXX

If the load function dynamically applies for memory , the unload function must release the memory

If the load function applies for hardware resources (interrupts , DMA channels , I/O ports, etc.), the unload function must release these resources

If the load function turns on the hardware , the unload function generally needs to turn off the hardware

General unloading function , clear / release code resource , best to keep the initialization function in reverse order , the resources that is the final application , the first release . Although this is not mandatory .

Module parameters

We can use " module_param (parameter name , parameter type , parameter read / write permission) " to define a parameter for the module , or use module_param_array (array name , array type , array length , parameter read / write permission).

For example, the following code defines an integer parameter and a character pointer parameter :

```
#include <linux/moduleparam.h>
```

```
static char *whom = "world";
static int howmany = 1;
module_param( howmany, int, S_IRUGO);
module_param( whom, charp, S_IRUGO);
```

```
static int int_array[6] = {1, 2, 3, 4, 5, 6};
static int int_array_num;
module_param_array( int_array, int, &int_array_num, S_IRUGO);
```

Explain the key points of the module parameters :

moduleparam.h	The module_param macro is defined in the header file
module_param	Macro , used to indicate module parameters
Parameter Type	The type of parameters can be : bool / invbool: Boolean value / Istanbul Anti short , ushort, int, uint, long, ulong charp: character pointer

Array length	<p>Version 2.6.0 ~ 2.6.10 , you need to assign the array long variable name to " array length "</p> <p>After version 2.6.10 , you need to assign the pointer of the array long variable to the " tree leader"</p> <p>When the number of stored head loading module , the actual number of arguments passed to the array . When the number of array elements do not need to actually input , may be provided , " the number of head " is NULL</p>
Read / write permissions	<p>Read and write permissions macros defined in <linux / stat.h> in .</p> <p>After the module is loaded , in / sys / module / directory to this directory named module name appears . If the " read / write permissions " is not 0, the module directory under the name , there will be a parameters directory ,The parameters directory contains a series of files named with parameter names . The read and write permissions of these files are specified by the " read / write permissions " in module_param .</p> <p>Of course , if the module parameter " read / write permission " is 0, the parameter will not appear in sysfs . If the module does not have any module parameters or all parameters " read / write permission " are 0, the parameters directory will not appear .</p> <p>S_IRUGO: Anyone can read this parameter , but cannot modify it</p> <p>S_IRUGO S_IWUSR: anyone can read it , root user can modify</p>

When writing the kernel module code , a default value is specified for each module parameter . The default value can be modified by the command insmod or modprobe when loading the module , for example :

```
insmod hello.ko howmany=10 whom= " Mom " int_array=3 ,4
```

modprobe can also read parameter values from its configuration file (/etc/modprob.conf)

When the module is loaded , if the module parameters " read / write access to " set right , we can also sysfs to retrieve or modify the parameter values . Note , if a parameter through sysfs is modified , the modification action will take effect immediately , but the kernel does not Any mechanism to notify the module that a parameter has been modified .

Kernel symbol table

The " /proc/kallsyms " file corresponds to the symbol table exported by the kernel , which records the symbol and the memory address where the symbol is located .

The kernel module can call symbols (also called functions) exported by the kernel , and the kernel module itself can also export symbols for use by other modules .

You can use the following macro to export symbols to the kernel symbol table :

```
EXPORT_SYMBOL ( name);
```

```
EXPORT_SYMBOL_GPL ( name);
```

The symbols exported by EXPORT_SYMBOL_GPL can only be used if GPL LICENSES is declared in the kernel module code

Module usage count

In Linux 2.6 in , module typically handled automatically by the counting device model system , the main purpose is to prevent the device being used , uninstall the device corresponding to the module .

For example, we wrote a kernel module , register a character device in the module code which , when the system is started using the character device , the system will automatically call try_module_get (dev-> owner) to increase the use of this device corresponds counting module , when not when using this device , the kernel uses

module_put (dev-> owner) to reduce the usage count corresponding to the device modules . when the usage count is not 0 when , not unload the module .

Here is only a brief introduction , and I will elaborate on it later .

Version dependency

The version of the Linux kernel is constantly changing . For different versions , the exported kernel symbols may be different , and the parameters of the kernel symbols may also be different . In addition to these , there may be other differences .

If we plan to write a kernel module so that it can be compiled on different versions of the kernel , we need to use some #ifdef to organize our own code .

The version of the kernel will be defined in linux/version.h ,This header file is automatically included in the linux / module.h in , the header defines the following macros :

UTS_RELEASE	A string describing the kernel version , such as " 2.6.10 "
LINUX_VERSION_CODE	Binary data , each part of the version release number corresponds to one byte . For example : 2.6.10 corresponds to LINUX_VERSION_CODE is 132618 (ie 0x02060a)
KERNEL_VERSION (major, minor, release)	Used to create a binary version number , for example , KERNEL_VERSION(2, 6, 10) represents 132618. This macro is very useful when we need to compare the current version with a known version

Through the above macros , most version dependency issues can be handled . But don't use #ifdef conditional statements in the kernel code to mess up the entire driver code . The best solution is to put all relevant pre-conditional statements Stored in a specific header file .

It depends on the particular version of the code should be hidden in the bottom of the details , higher-level code can call these functions directly , without attention to the underlying details . Code written in this way easy to read , but also more robust .

Platform dependence

Each CPU has its own unique characteristics , and kernel designers can make full use of these characteristics to achieve the optimal performance of the target file on the target platform .

This mainly refers to the kernel code according to different needs , the CPU designate certain special purpose registers , and thus play a CPU power .

This piece is not elaborated at present , but I will discuss it in detail later .

If this happens , we need to remember is , these low-level details hide , to improve the robustness of the program .

4. Drive design guidelines

Code path and description

File	Comment
#include <linux/init.h>	<p>Macros that indicate the initialization and clearing functions of the module</p> <pre>module_init(init_function); module_exit(cleanup_function)</pre> <p>Mark functions or data that are only used in the initialization / clearing phase</p> <pre>__init, __initdata __exit, __exitdata</pre>
#include <linux/module.h>	<p>Header files that must be included in the module source code</p> <pre>MODULE_LICENSE MODULE_AUTHOR MODULE_DESCRIPTION MODULE_VERSION MODULE_DEVICE_TABLE MODULE_ALIAS</pre>
#include <linux/sched.h>	<p>Contains definitions of most of the kernel APIs used by the driver , including sleep functions and various variable declarations</p> <pre>struct task_struct *current</pre> <p>current represents the current process</p> <pre>current->pid current->comm</pre> <p>The current process ID and command name</p>
#include <linux/version.h>	<p>This header file is automatically included in linux / module.h in</p> <p>Header file for kernel version information</p> <pre>UTS_RELEASE LINUX_VERSION_CODE KERNEL_VERSION</pre>
#include <linux/export.h>	<p>This header file is automatically included in linux / module.h in</p> <pre>EXPORT_SYMBOL(name); EXPORT_SYMBOL_GPL(name);</pre>
#include <linux/moduleparam.h>	<p>This header file is automatically included in linux / module.h in</p> <pre>module_param(variable, type, perm) module_param_array (variable, type, num, perm)</pre>
#include <linux/kernel.h>	<pre>int printk(const char * fmt, ...);</pre>
vermagic.o	<p>An object file in the kernel source tree , which describes the construction environment of the module</p>
/sys/module	<p>The directory containing the information of the currently loaded module , it is a directory</p>
/proc/modules	<p>This file is an early usage , it is a single file</p>
/proc/kallsyms	<p>Kernel symbol table file</p>

Module loading / unloading / information related tools

Tools	Comment
-------	---------

insmod	Load module , can pass module parameters
rmmod	Uninstall the module
modprobe	<p>Load the module , and at the same time process the dependencies between the modules . Module parameters can be passed</p> <p>You need to be compiled modules into / lib / module / `uname -r` down , and with the depmod command to update modules.dep file</p> <p>Modprobe with the module followed by the module name , rather than the module file name</p>
lsmod	View loaded module information
modinfo	<p>modinfo hello.ko can view kernel information , including</p> <p>Module author</p> <p>Module description</p> <p>Parameters supported by the module</p> <p>And vermagic</p>

Kernel module writing guide

1. Include related header files
2. Write module loading function (required)
3. Write module uninstall function (required)
4. Module LICENSES statement (required)
5. Other information such as module author , MODULE_AUTHOR, etc. (recommended)
6. Module parameters (if the module requires parameters)
7. Module export symbols (if the module needs to export symbols)
8. Consider module-related version dependencies / platform dependencies (if you need to consider)